Advanced Data Structures and Algorithm Analysis

丁尧相
浙江大学

Fall & Winter 2025
Lecture 2

Balanced Search Trees (II)

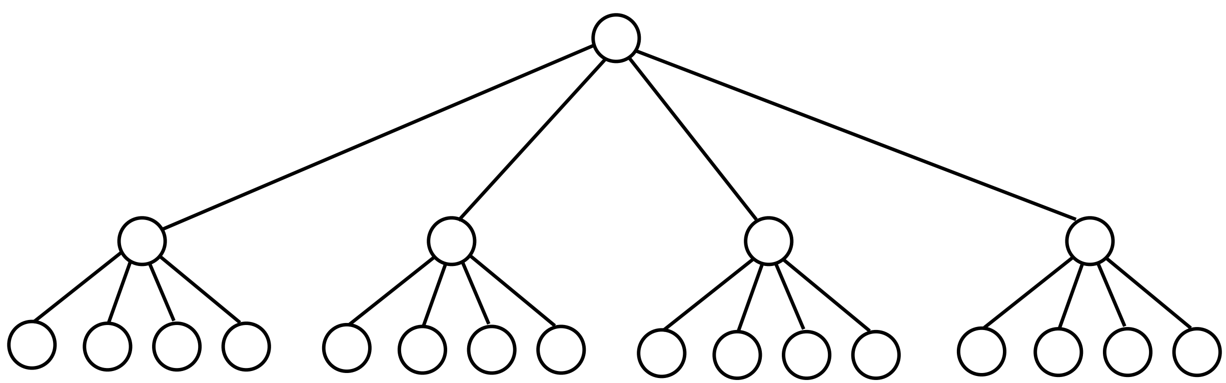
- Red-black trees
- B & B+ trees
- Take-home messages

Balanced Search Trees (II)

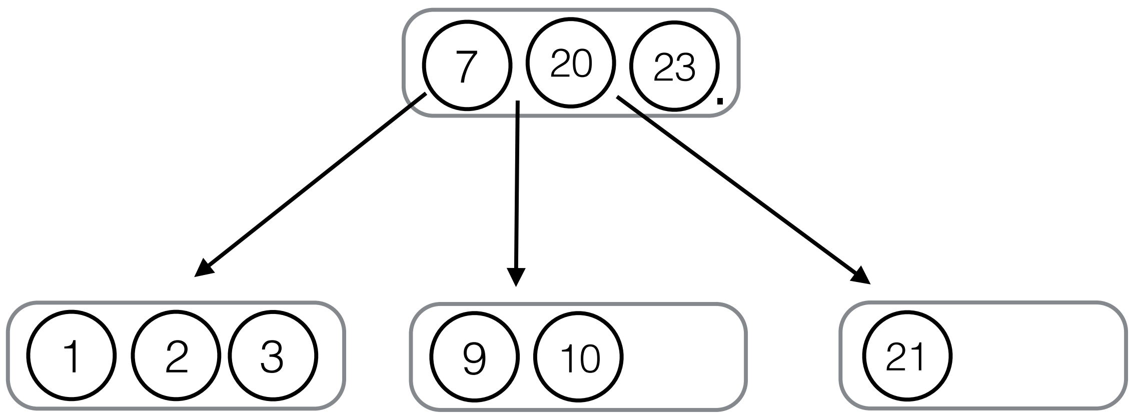
- Red-black trees
- B & B+ trees
- Take-home messages

Generalizing Balanced BSTs

- AVL trees and Splay trees are good for searching due to the balancing condition. But if we want fewer rotation operations when inserting and deleting:
    - Sacrifice a little searching cost
    - Relax balancing condition



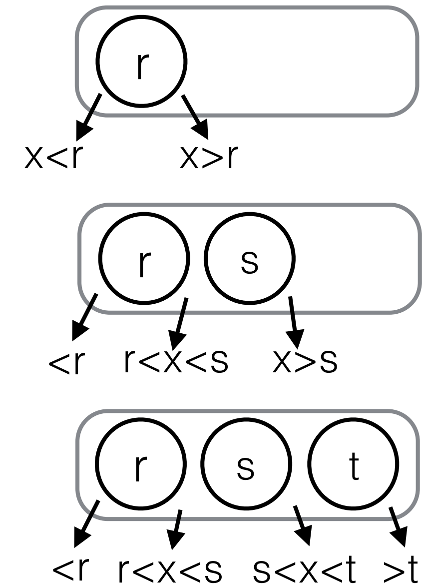
M-ary Search Trees



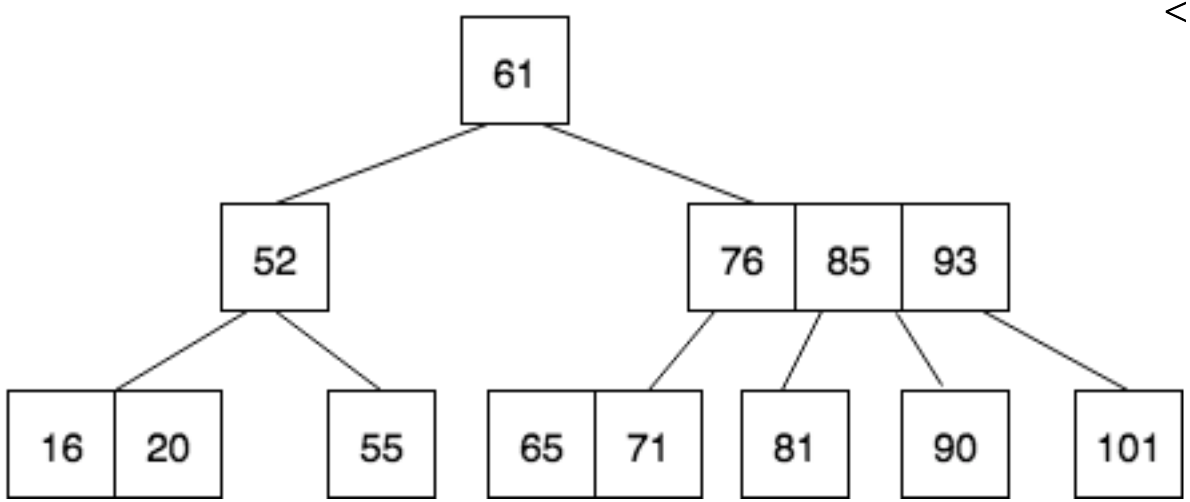
4-ary search tree:
Nodes have 1,2, or 3 data items and 0 to 4 children.

2-3-4 Trees (B-Tree Version)

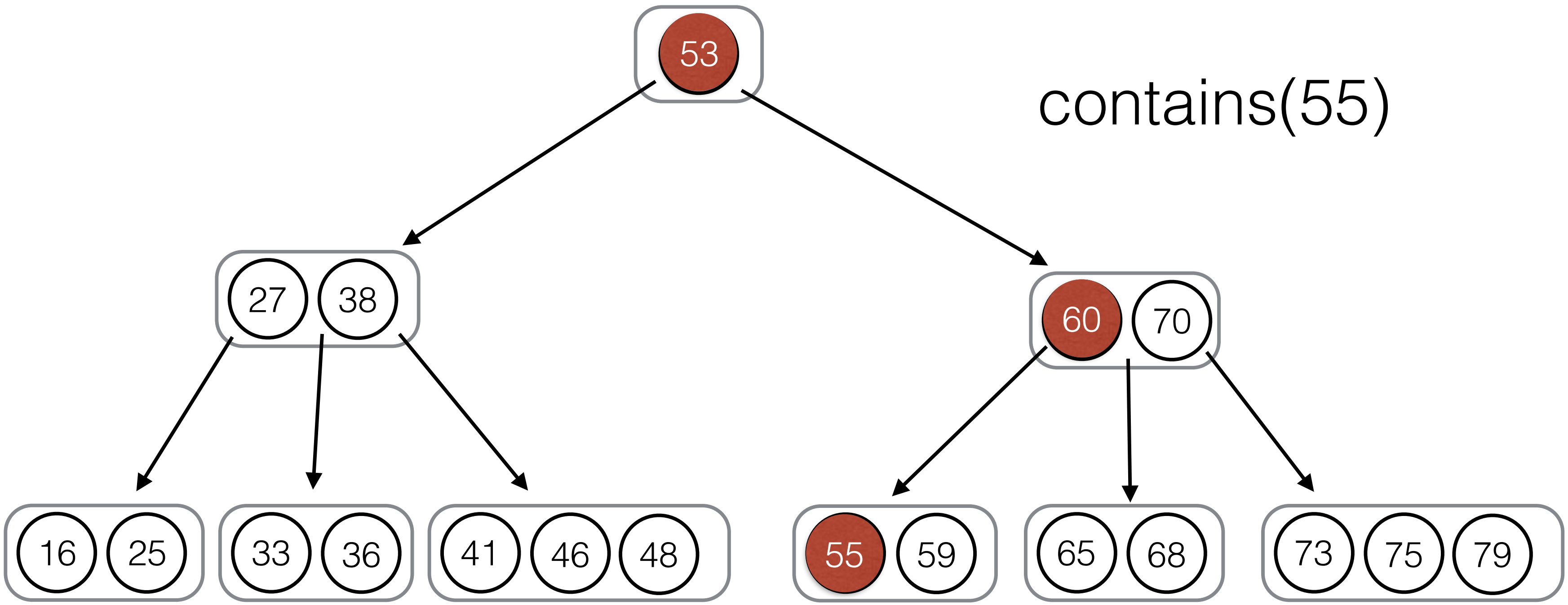
- A 2-3-4 tree is a balanced 4-Ary search tree.
- Three types of internal nodes:



- Balance condition:
  - All leaves have the same depth.



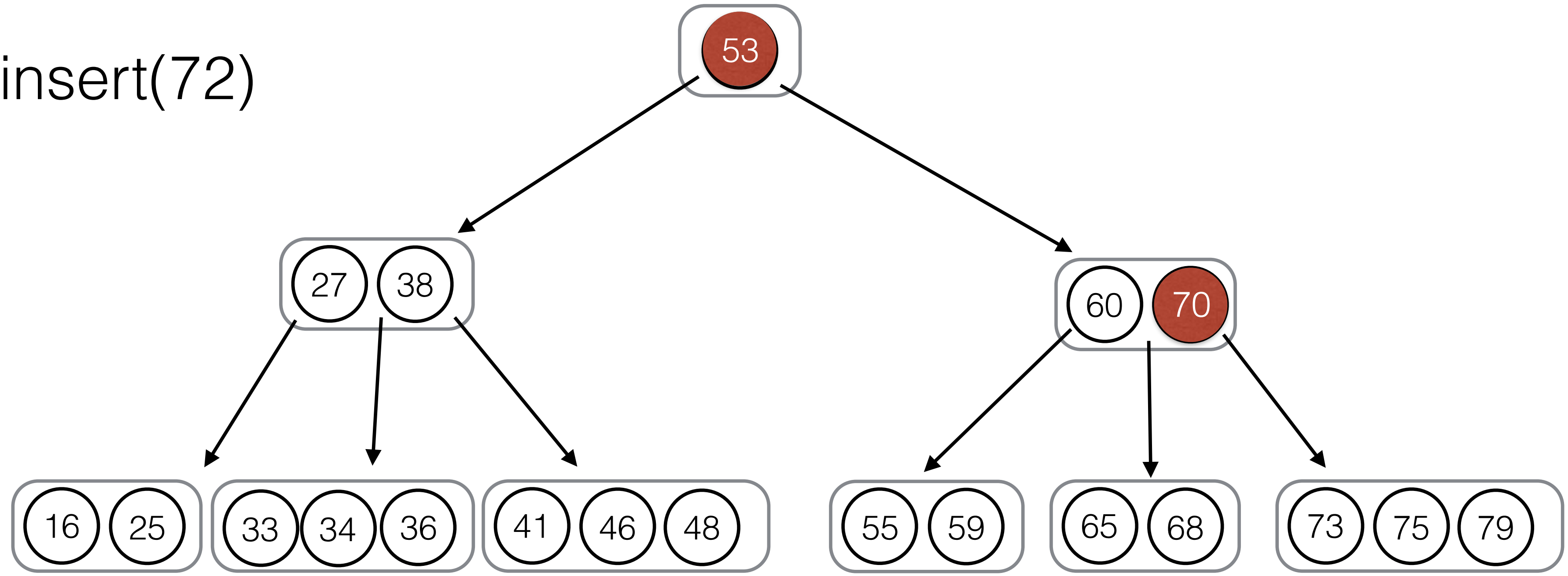
Searching



Linear search in each node. O(3d) time cost.

Insertion

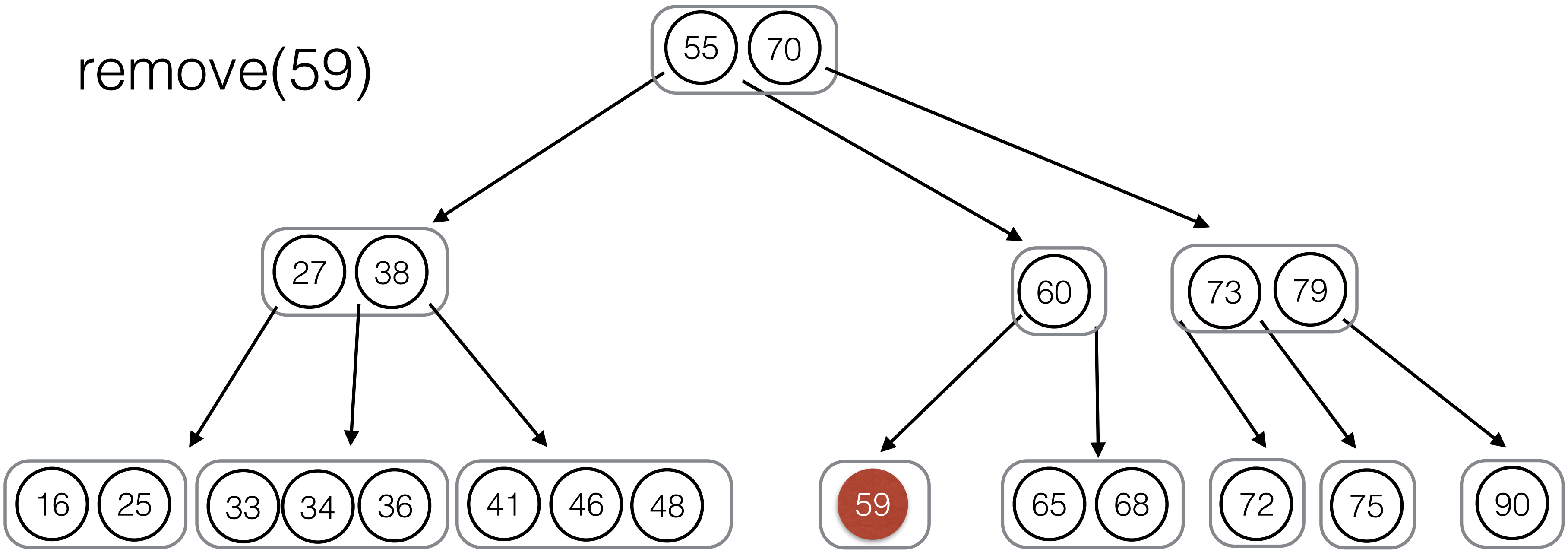
insert(72)



The insertion happens on the leaves.
When the leaf is full, splitting needs to be done.

Deletion

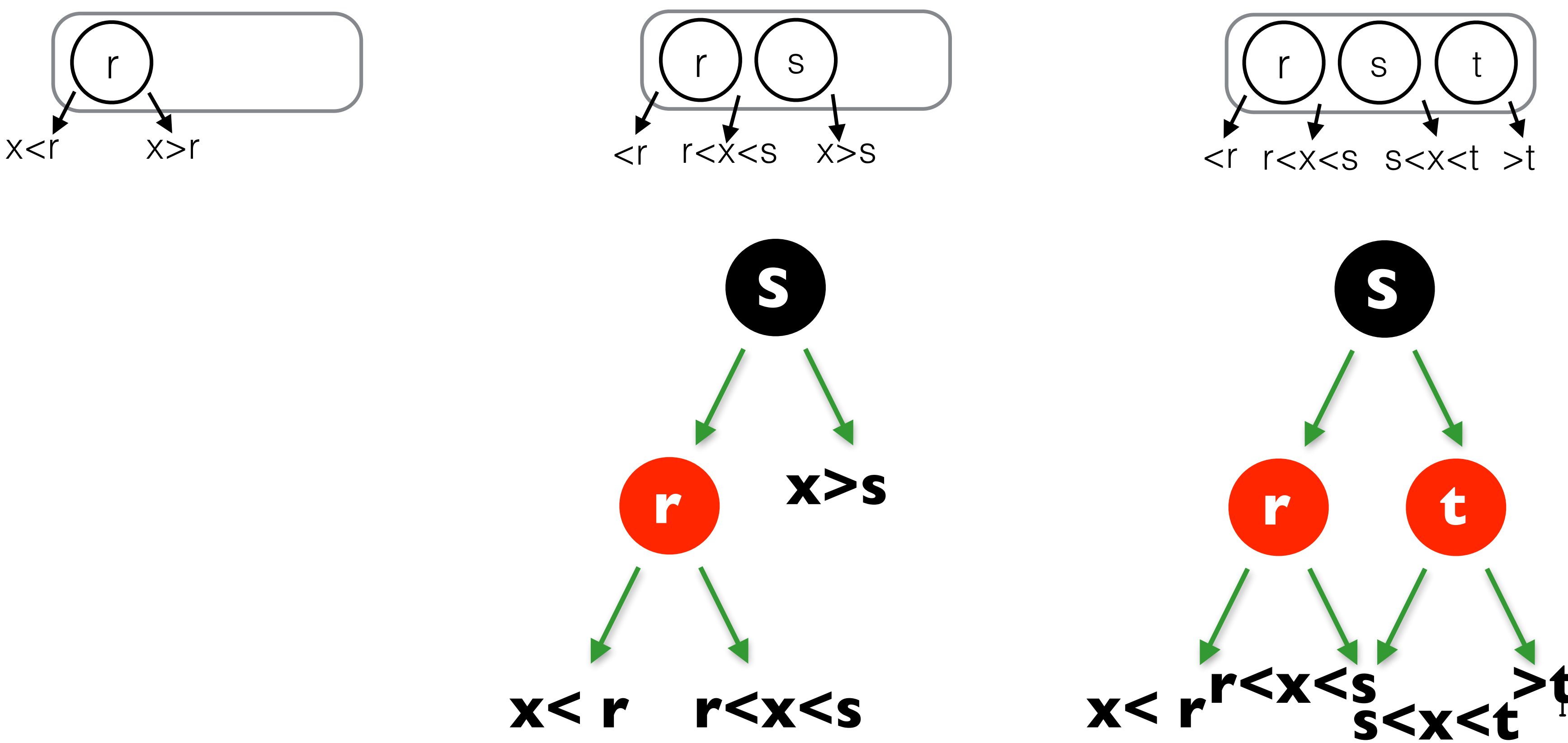
remove(59)



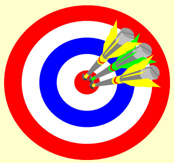
Deletions can make the nodes not satisfy the minimum number of keys (e.g. 2).
Needs further manipulation (combine)
Can we make insertion and deletion easy with binary search tree?

Red-Black Trees

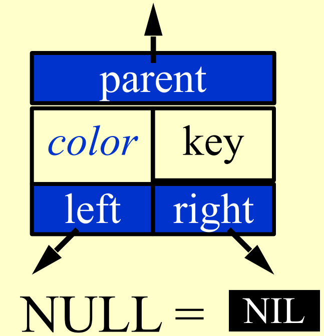
- Reduce 2-3-4 trees to BSTs:
  - The key is to transform 3- and 4- nodes into 2-nodes:



Red-Black Trees



Target : Balanced binary search tree



【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

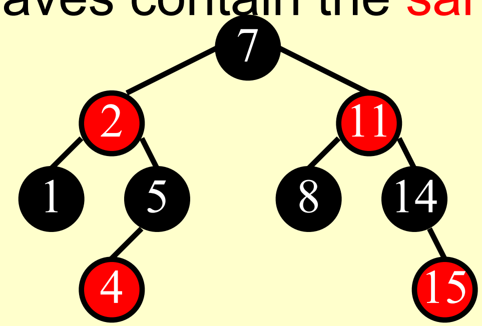
(1) Every node is either **red** or **black**.

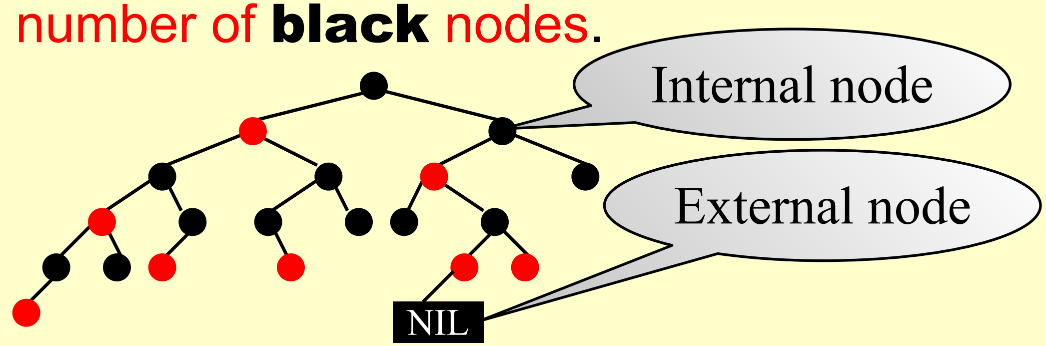
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

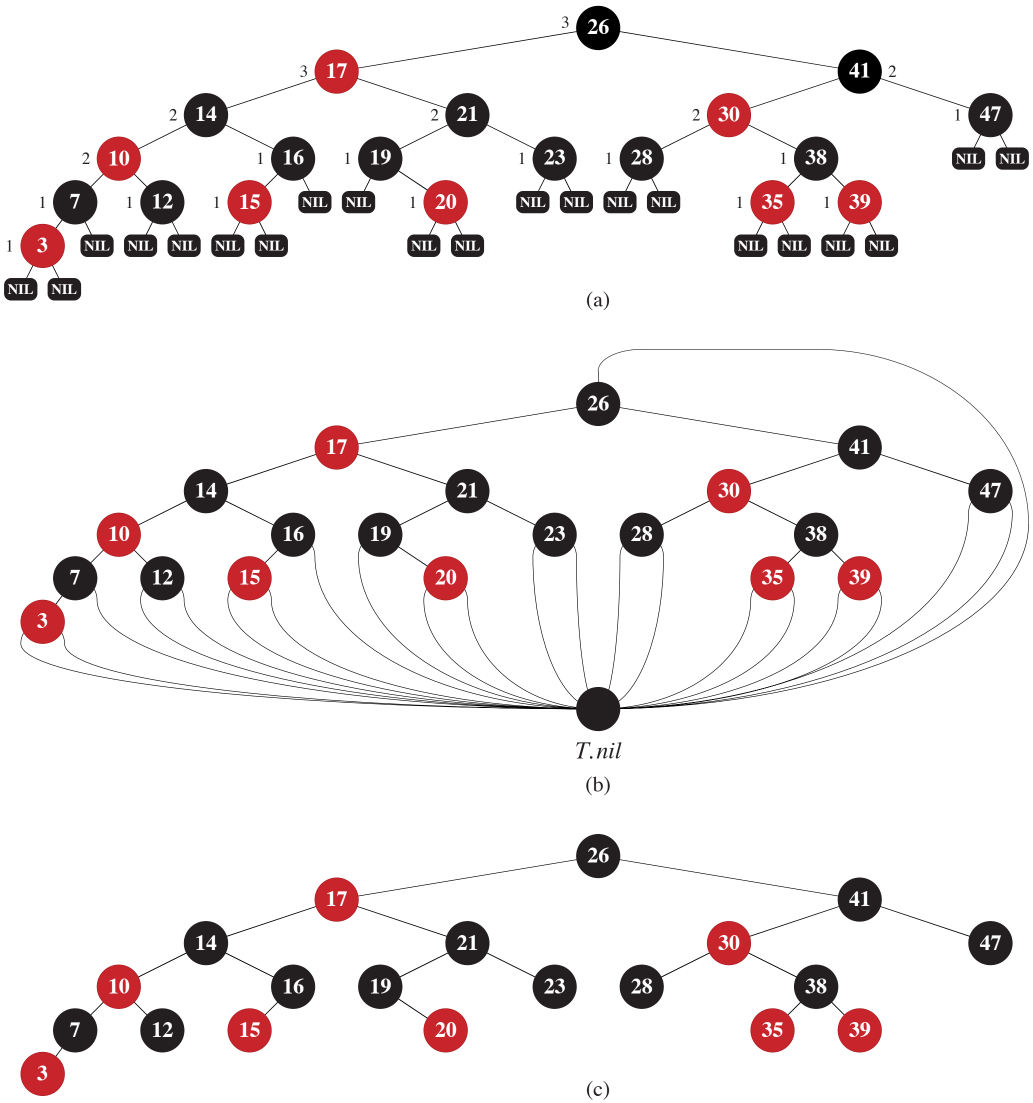
(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.





Red-Black Trees



How balanced are red-black trees?

【Definition】 The black-height of any node x, denoted by bh(x), is the number of **black** nodes on any simple path from x (x not included) down to a l

Number of internal nodes in the subtree rooted at x

【Lemma】 A red-b ... ght at most $2\ln(N+1)$.

Proof: ① For any node x, $\text{sizeof}(x) \geq 2^{\text{bh}(x)} - 1$. Prove by induction.

If $h(x) = 0$, x is NULL $\Rightarrow$ $\text{sizeof}(x) = 2^0 - 1 = 0$

Suppose it is true for all x with $h(x) \leq k$.

For x with $h(x) = k + 1$, $\text{bh}(child) = ?$ $\text{bh}(x)$ or $\text{bh}(x) - 1$

Since $h(child) \leq k$, $\text{sizeof}(child) \geq 2^{\text{bh}(child)} - 1 \geq 2^{\text{bh}(x) - 1} - 1$

Hence $\text{sizeof}(x) = 1 + 2\text{sizeof}(child) \geq 2^{\text{bh}(x)} - 1$

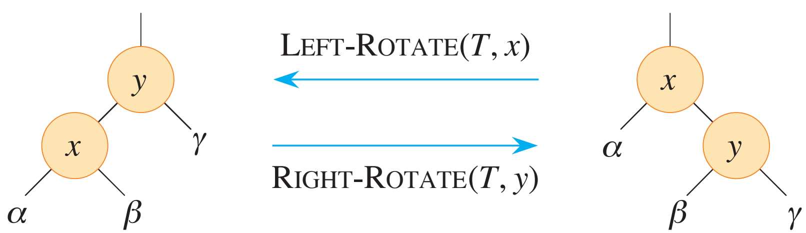
② $\text{bh}(Tree) \geq h(Tree) / 2$?

Since for every red node, both of its children must be black, hence on any simple path from *root* to a leaf, at least half the nodes (*root* not included) must be black.

$$\text{Sizeof}(root) = N \geq 2^{\text{bh}(Tree)} - 1 \geq 2^{h/2} - 1$$

Tree Insertion and Deletion

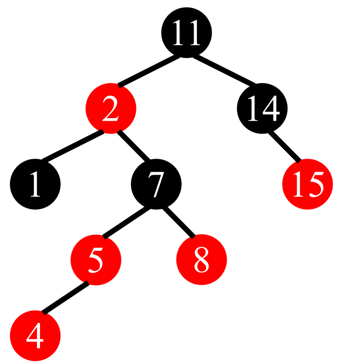
- Similar with AVL and splay trees, the rotations are usually required when insertion and deletion lead to the violation of tree properties.




```
LEFT-ROTATE(T, x)
  y = x.right
  x.right = y.left          // turn y's left subtree into x's right subtree
  if y.left ≠ T.nil         // if y's left subtree is not empty ...
      y.left.p = x          // ... then x becomes the parent of the subtree's root
  y.p = x.p                 // x's parent becomes y's parent
  if x.p == T.nil           // if x was the root ...
      T.root = y            // ... then y becomes the root
  elseif x == x.p.left      // otherwise, if x was a left child ...
      x.p.left = y          // ... then y becomes a left child
  else x.p.right = y        // otherwise, x was a right child, and now y is
  y.left = x                // make x become y's left child
  x.p = y
```

Need to reduce the number of rotations

Insertion



always insert the new node as a red node on the bottom.

【Definition】A red-black tree is a binary search tree that satisfies the following *red-black properties*:

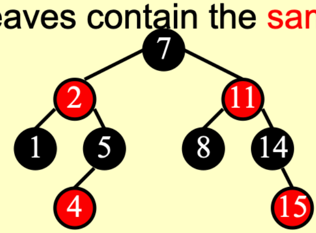
(1) Every node is either **red** or **black**.

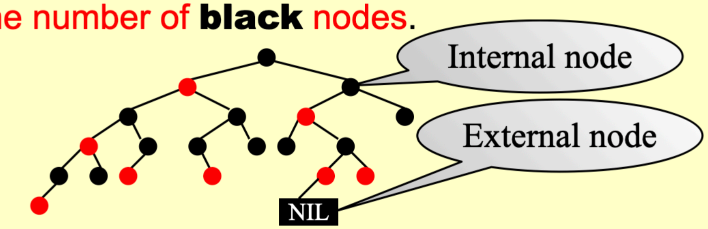
(2) The root is **black**.

(3) Every leaf (NIL) is **black**.

(4) If a node is **red**, then both its children are **black**.

(5) For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.



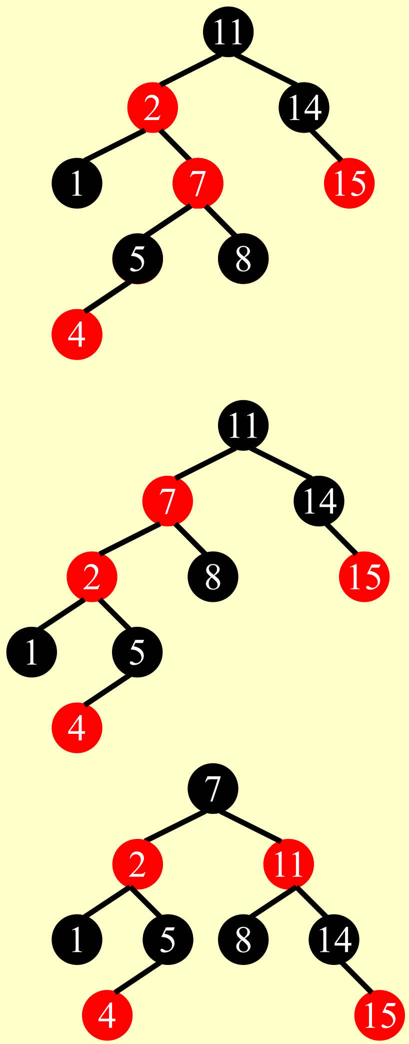


What properties can be violated?

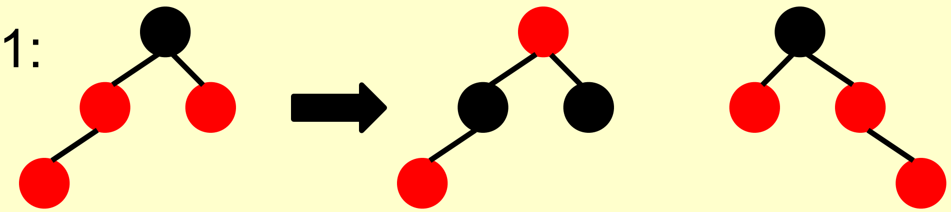
Only case 1 can repeat.
No more than 2 rotations.

Sketch of the idea: Insert & color red

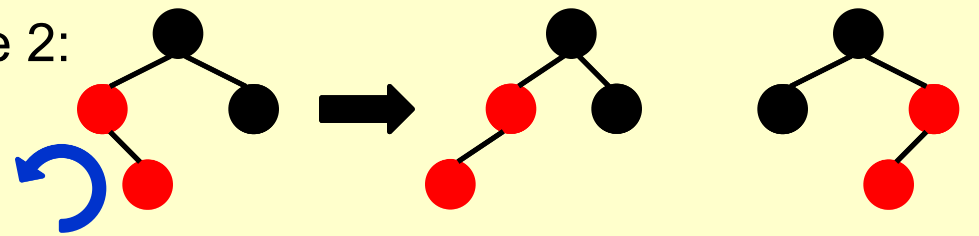
Symmetric



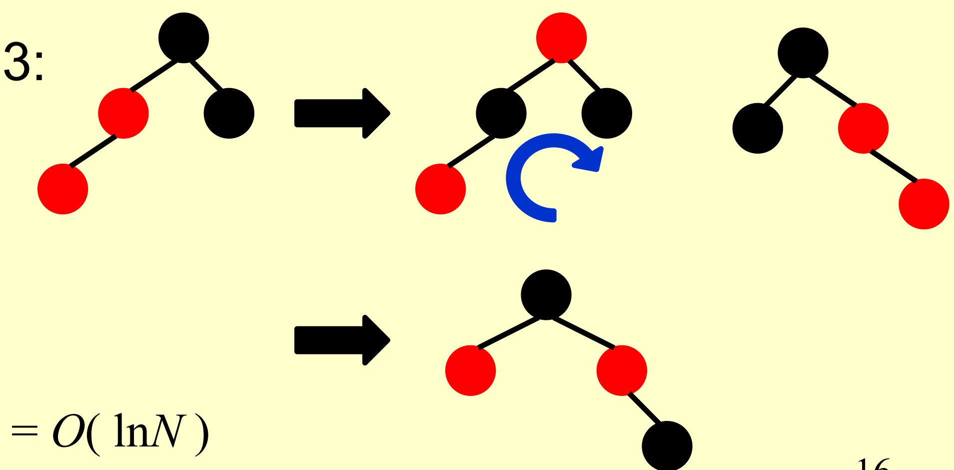
Case 1:



Case 2:



Case 3:



$T = O(h) = O(\ln N)$

Deletion in Normal BST

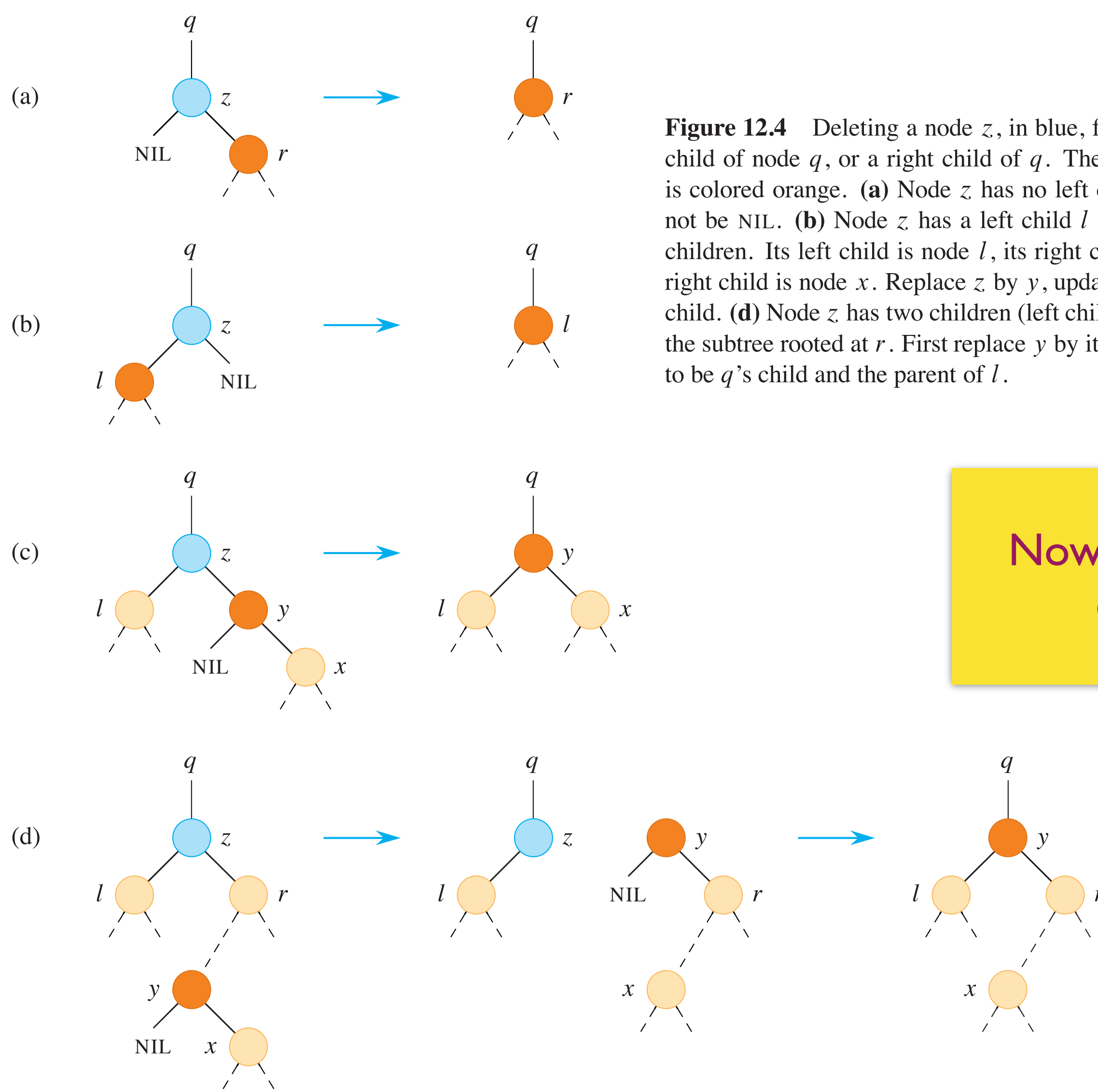
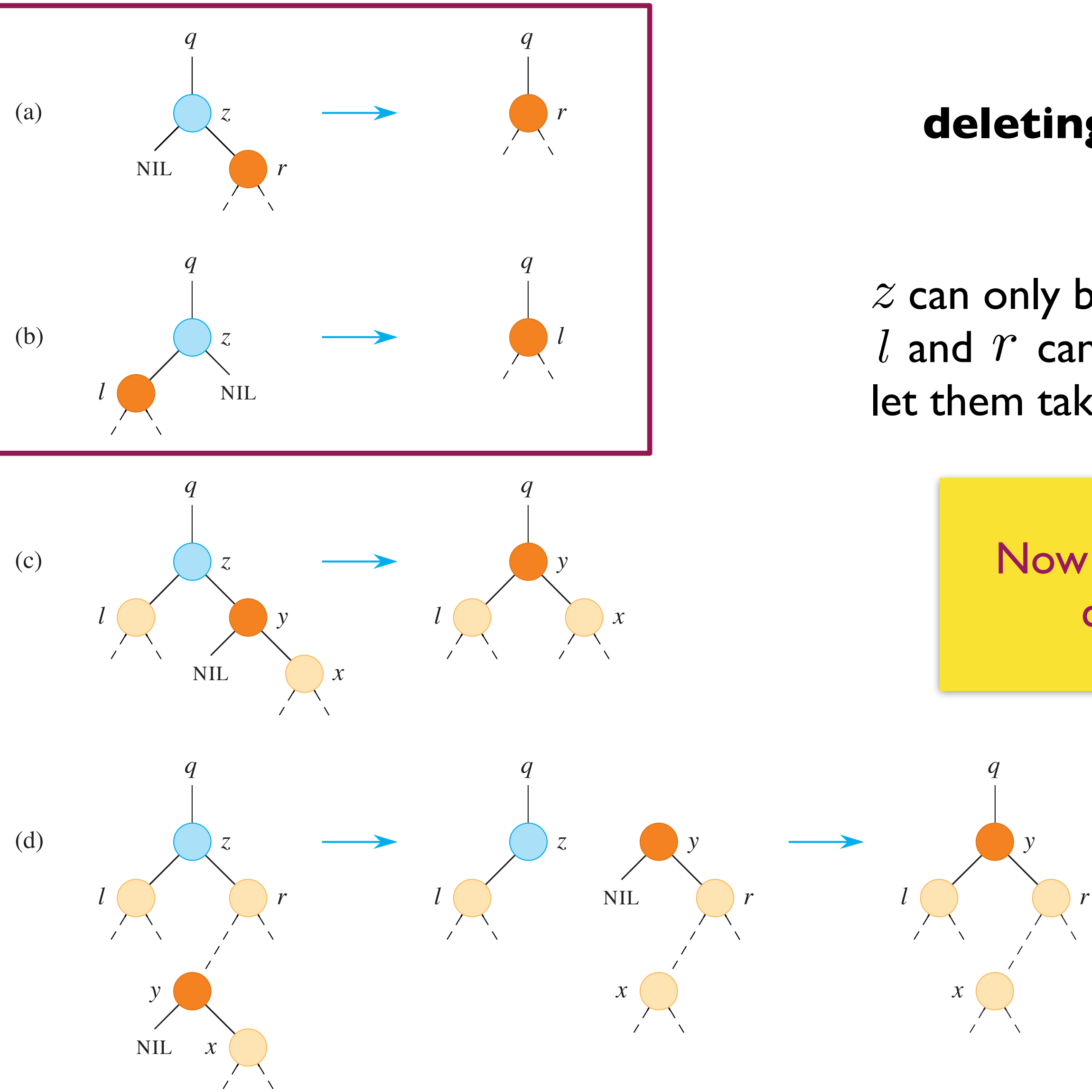


Figure 12.4 Deleting a node z, in blue, from a binary search tree. Node z may be the root, a left child of node q, or a right child of q. The node that will replace node z in its position in the tree is colored orange. **(a)** Node z has no left child. Replace z by its right child r, which may or may not be NIL. **(b)** Node z has a left child l but no right child. Replace z by l. **(c)** Node z has two children. Its left child is node l, its right child is its successor y (which has no left child), and y's right child is node x. Replace z by y, updating y's left child to become l, but leaving x as y's right child. **(d)** Node z has two children (left child l and right child r), and its successor $y \neq r$ lies within the subtree rooted at r. First replace y by its own right child x, and set y to be r's parent. Then set y to be q's child and the parent of l.

Now we need to consider color properties.

Deletion in Red-Black Tree

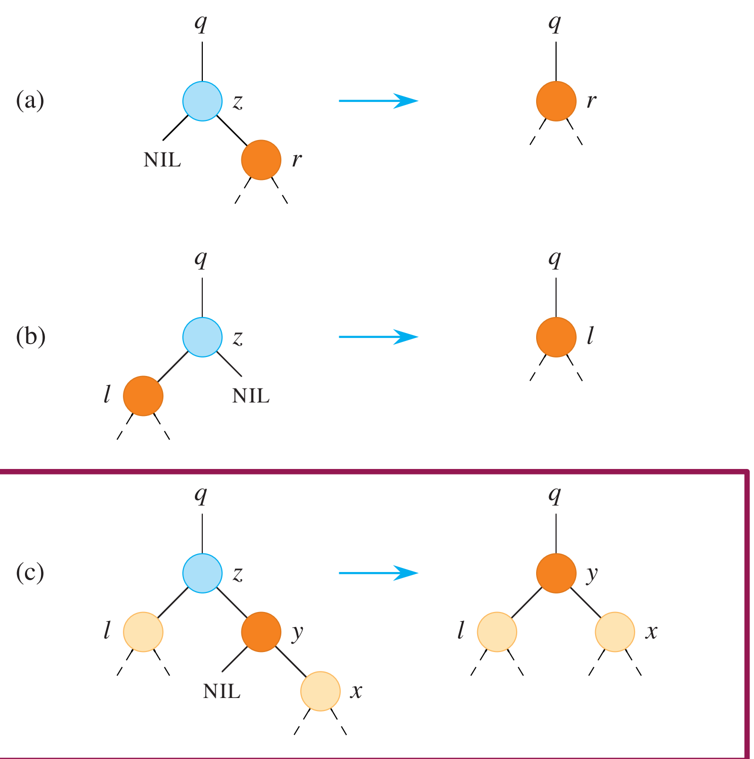


deleting a one-child node

z can only be black
l and r can only be red
let them take the place and color of z

Now we need to consider color properties.

Deletion in Red-Black Tree

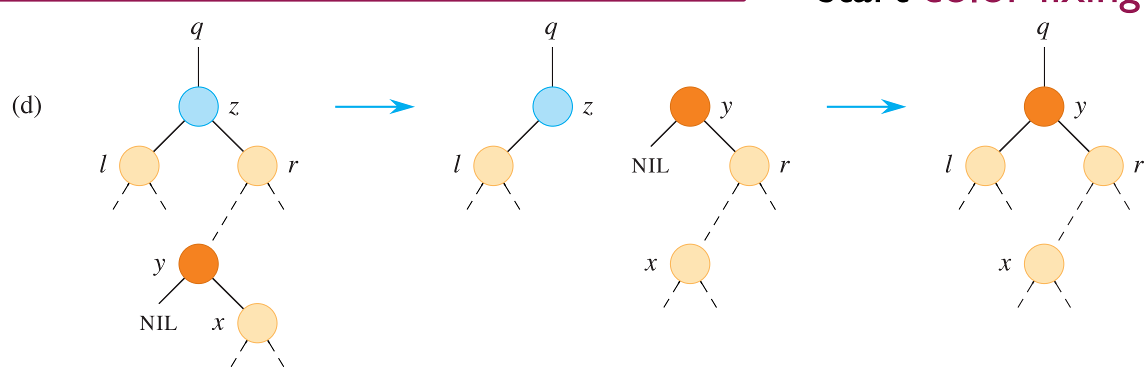


deleting a two-children node

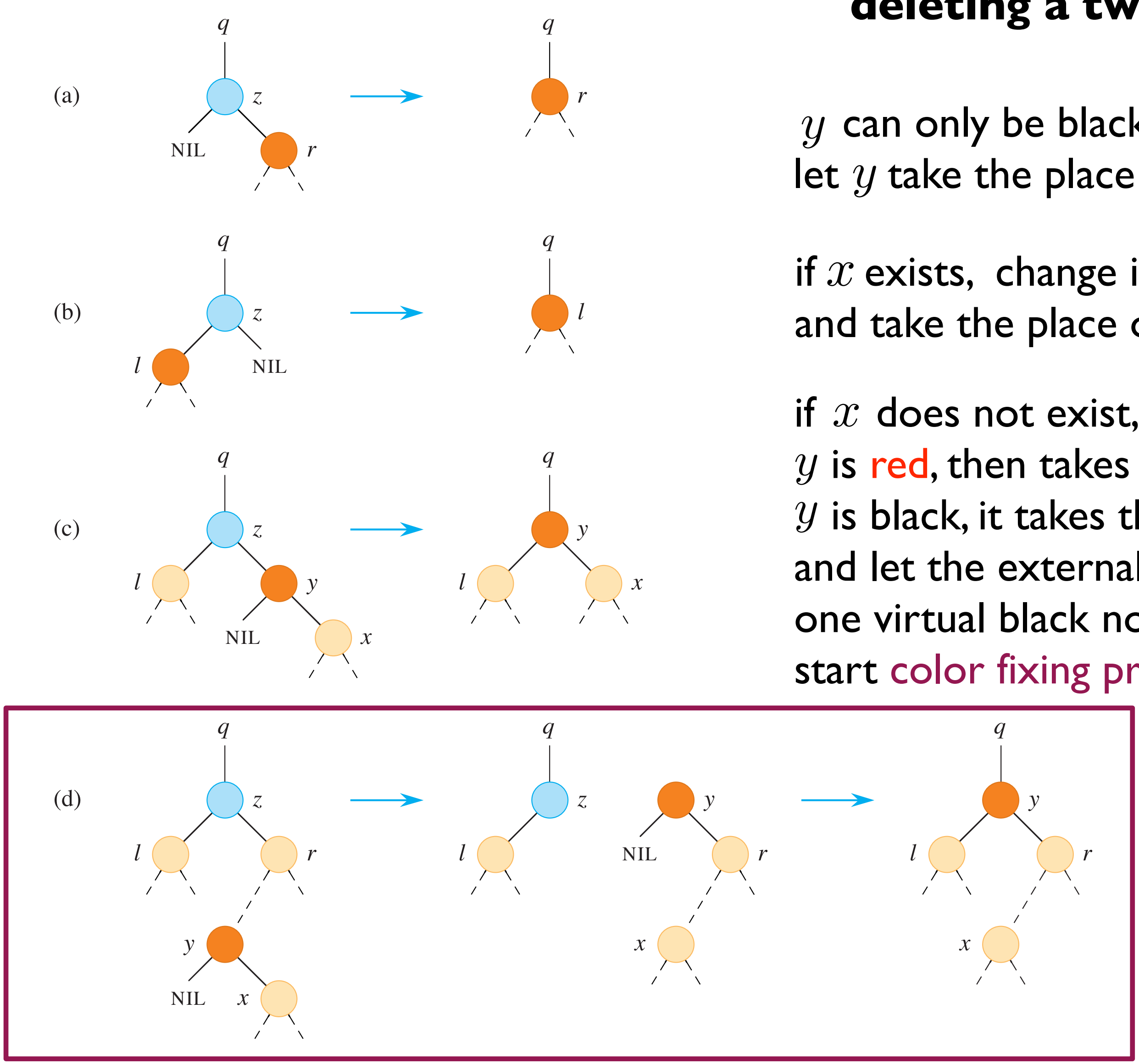
y can only be black, x can only be red
let y take the place and color of z

if x exists, change its color to black
and take the place of y

if x does not exist, y can be black or red
y is red, then takes the place and color of z
y is black, it takes the place of z
and let the external leaf node take its place:
one virtual black node included,
start color fixing process to cancel it out



Deletion in Red-Black Tree



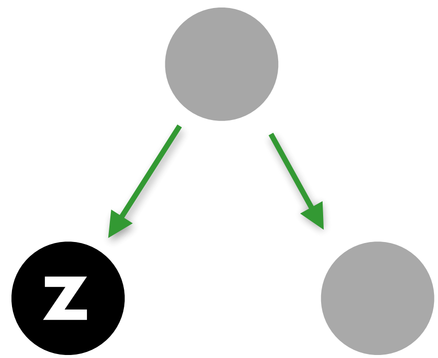
deleting a two-children node

y can only be black, x can only be red
let y take the place and color of z

if x exists, change its color to black
and take the place of y

if x does not exist, y can be black or red
y is red, then takes the place and color of z
y is black, it takes the place of z
and let the external leaf node take its place:
one virtual black node included,
start color fixing process to cancel it out

Deletion in Red-Black Tree



deleting a no-children (internal leaf) node

If color is red, direct delete.
If color is black, delete and let here be the virtual external leaf node and start color fixing process

- Delete a leaf node : Reset its parent link to NIL.
- Delete a degree 1 node : Replace the node by its single child.
- Delete a degree 2 node :

  ① Replace the node by the largest one ... the smallest one in its right subtree.

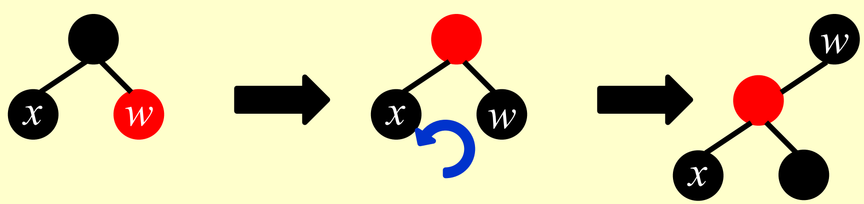
  ② Delete the replacing node from the subtree.

Adjust only if the node is black.

Keep the color

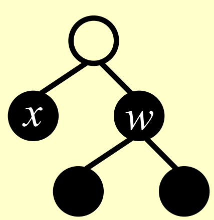
Must *add 1 black* to the path of the replacing node.

Case 1:

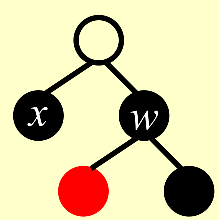


If x has no sibling then just color father black, or x must have a black sibling

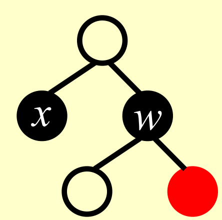
Case 2:



Case 3:

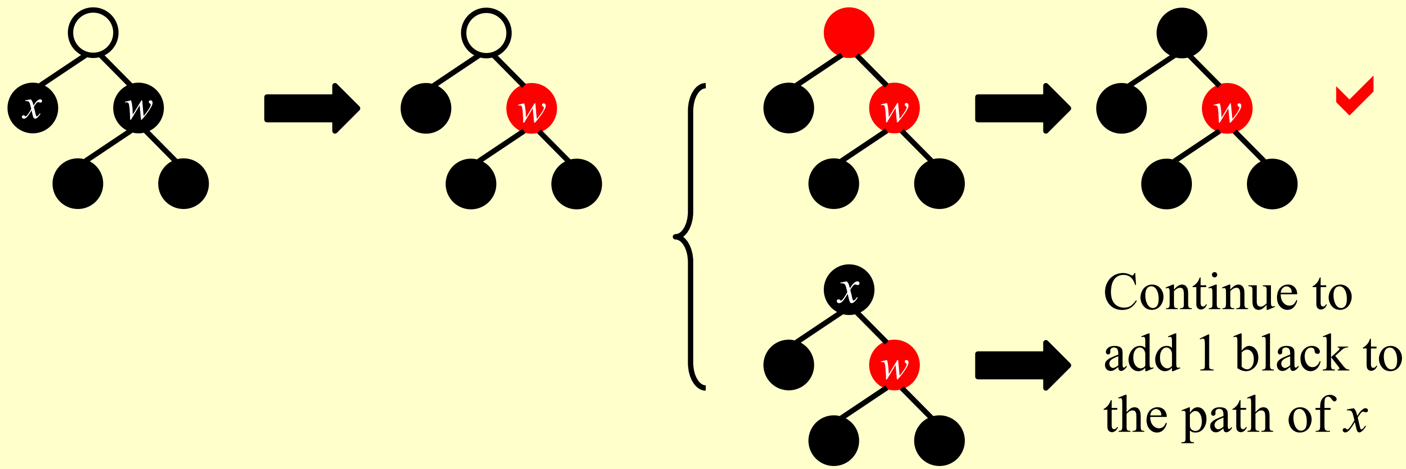


Case 4:

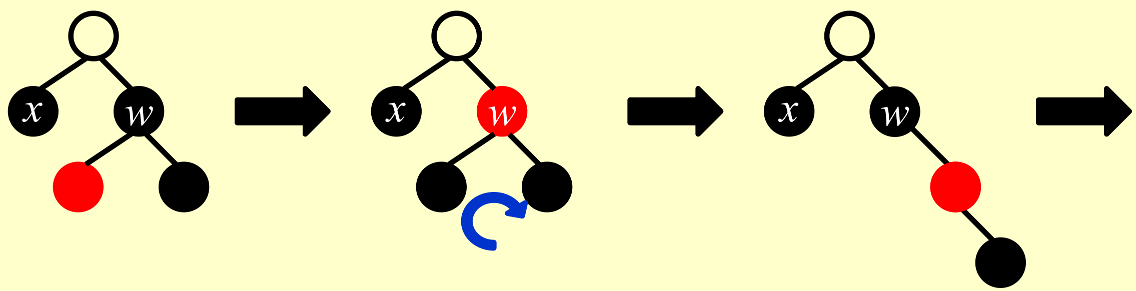


Cases are defined by sibling colors

Case 2:

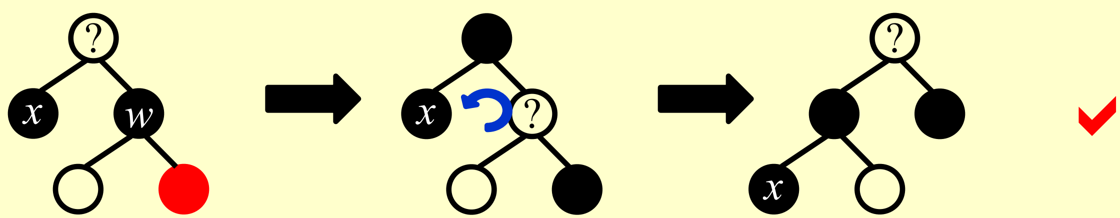


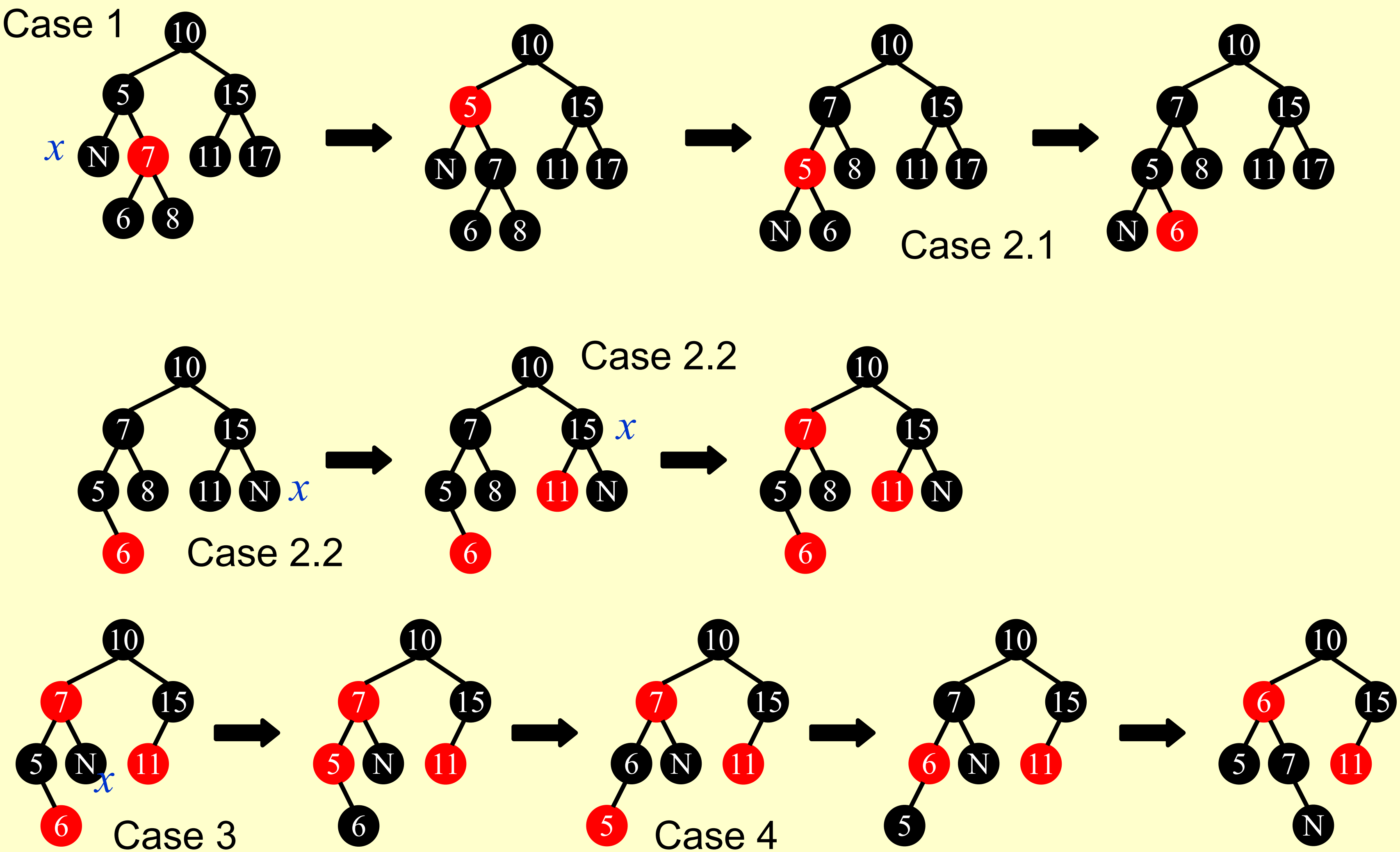
Case 3:



Case 4

Case 4:



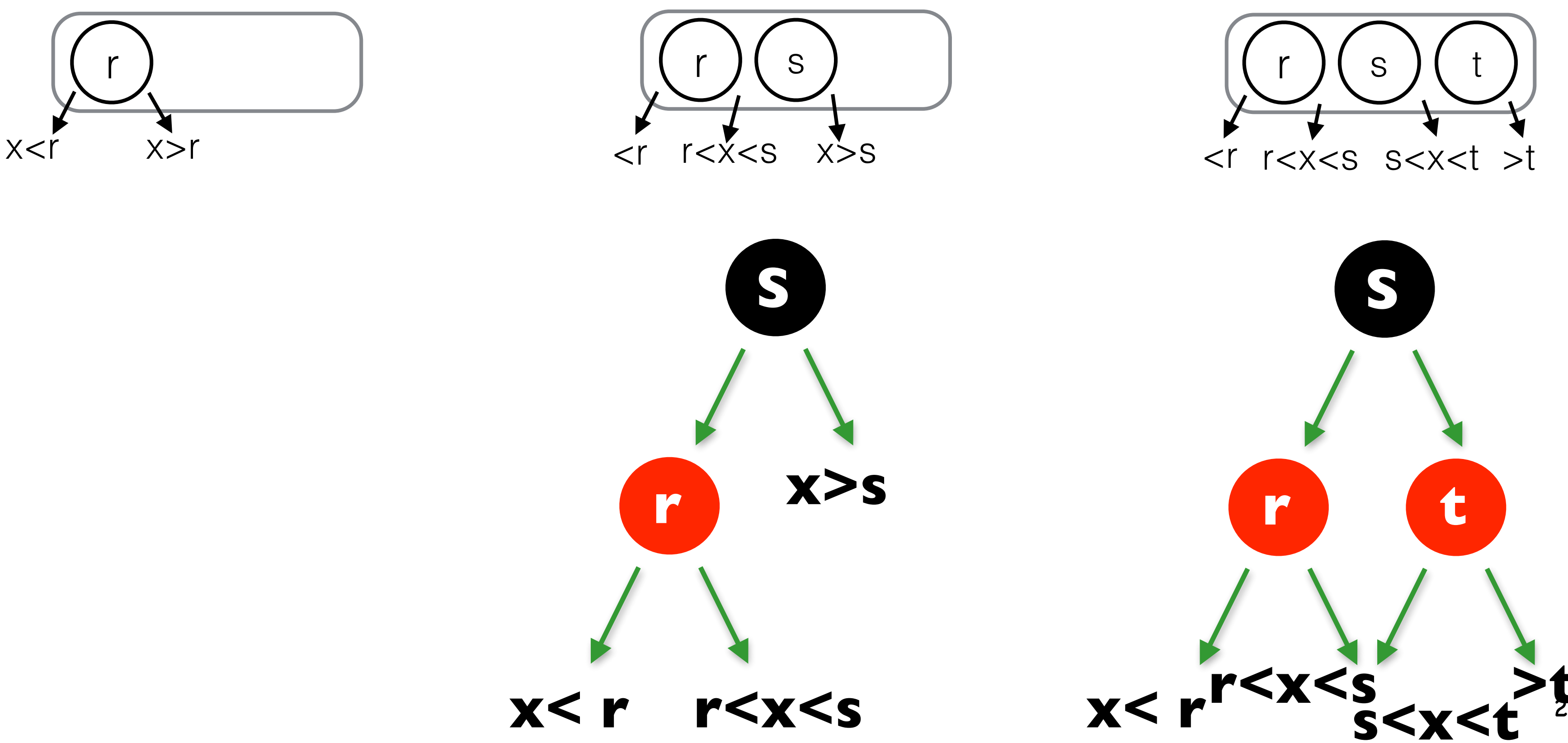

Case 1
Case 2.1
Case 2.2
Case 2.2
Case 3
Case 4

Number of *rotations*

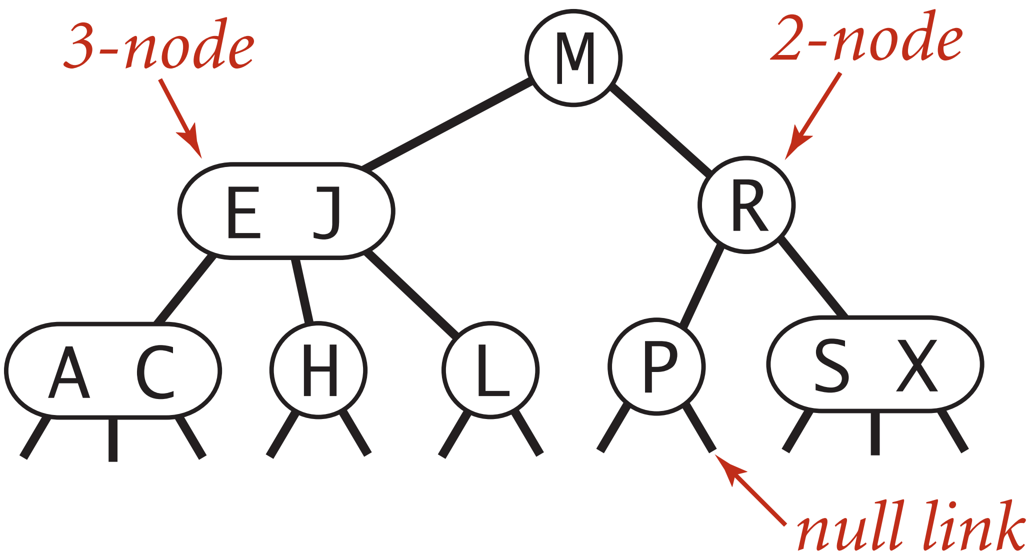
	AVL	Red-Black Tree
Insertion	≤ 2	≤ 2
Deletion	$O(\log N)$	≤ 3

Red-Black Trees

- Reduce 2-3-4 trees to BSTs:
    - The key is to transform 3- and 4- nodes into 2-nodes:

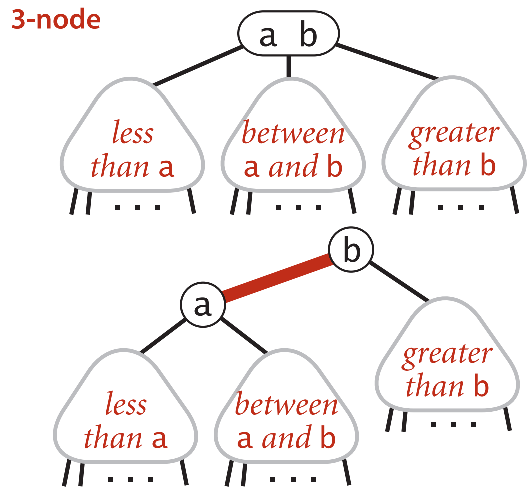


2-3 Trees

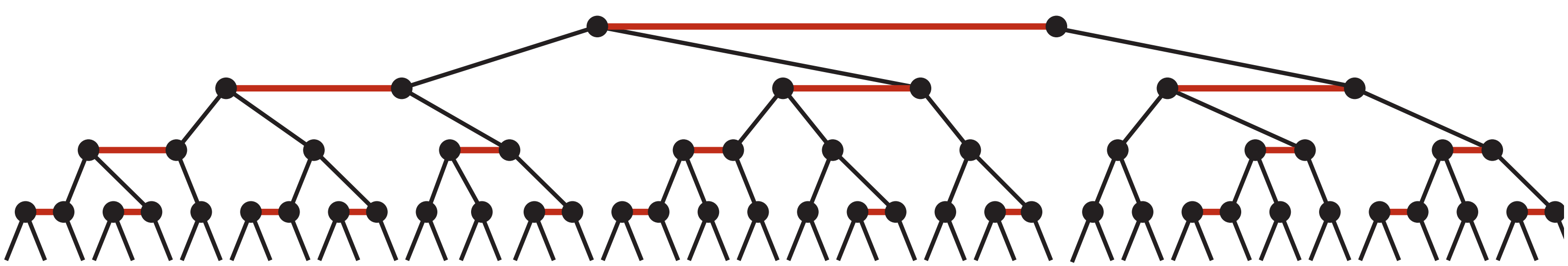


Transform into red-black tree?

Left-Leaning Red-Black Trees



Encoding a 3-node with two 2-nodes connected by a left-leaning red link



A red-black tree with horizontal red links is a 2-3 tree

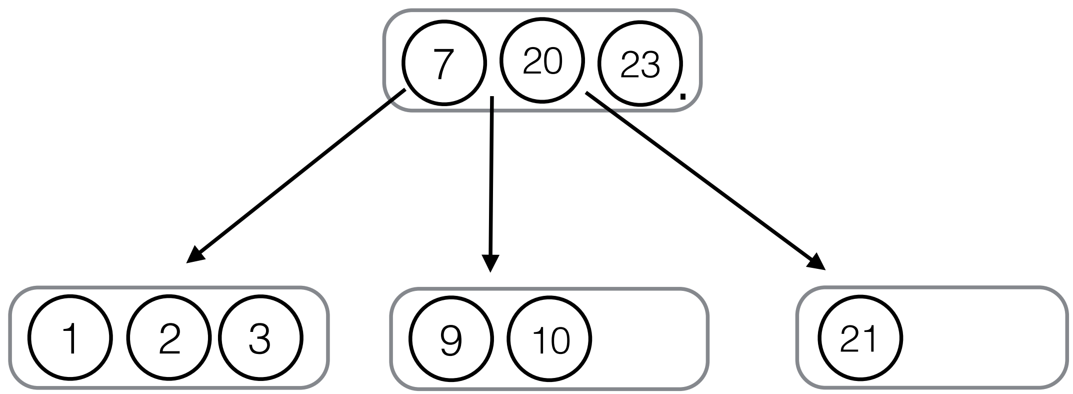
See [Sedgewick & Wayne] Chap. 3.3

Balanced Search Trees (II)

- Red-black trees
- B & B+ trees
- Take-home messages

M-ary Search Tree

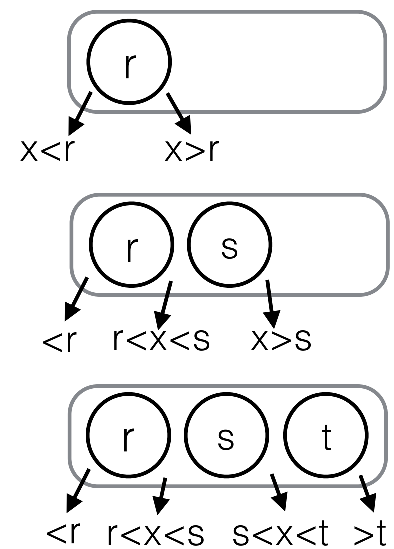
- We can generalize binary search trees to M-ary search trees.



4-ary search tree:
Nodes have 1,2, or 3 data items and 0 to 4 children.

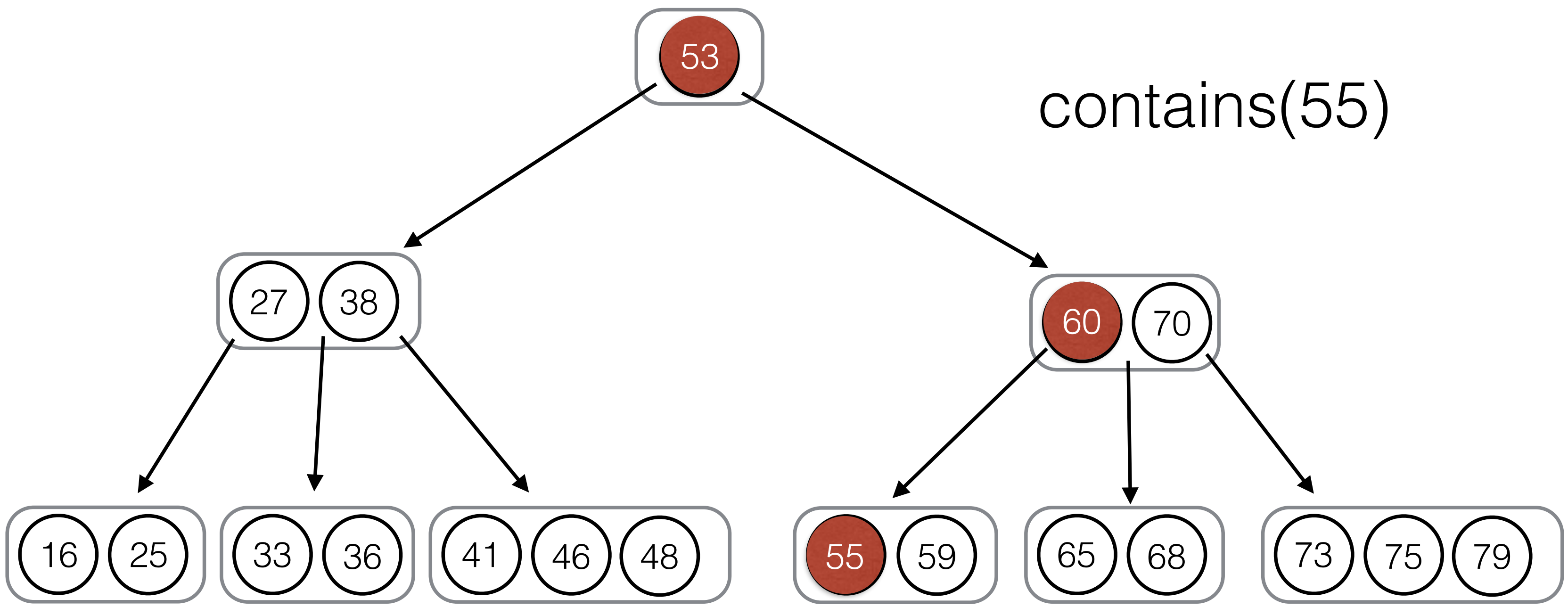
2-3-4 Trees

- A 2-3-4 Tree is a balanced 4-Ary search tree.
- Three types of internal nodes:
  - a 2-node has 1 item and 2 children.
  - a 3-node has 2 item and 3 children.
  - a 4-node has 3 item and 4 children.



- Balance condition:
  All leaves have the same depth.
  (height of the left and right subtree is always identical)

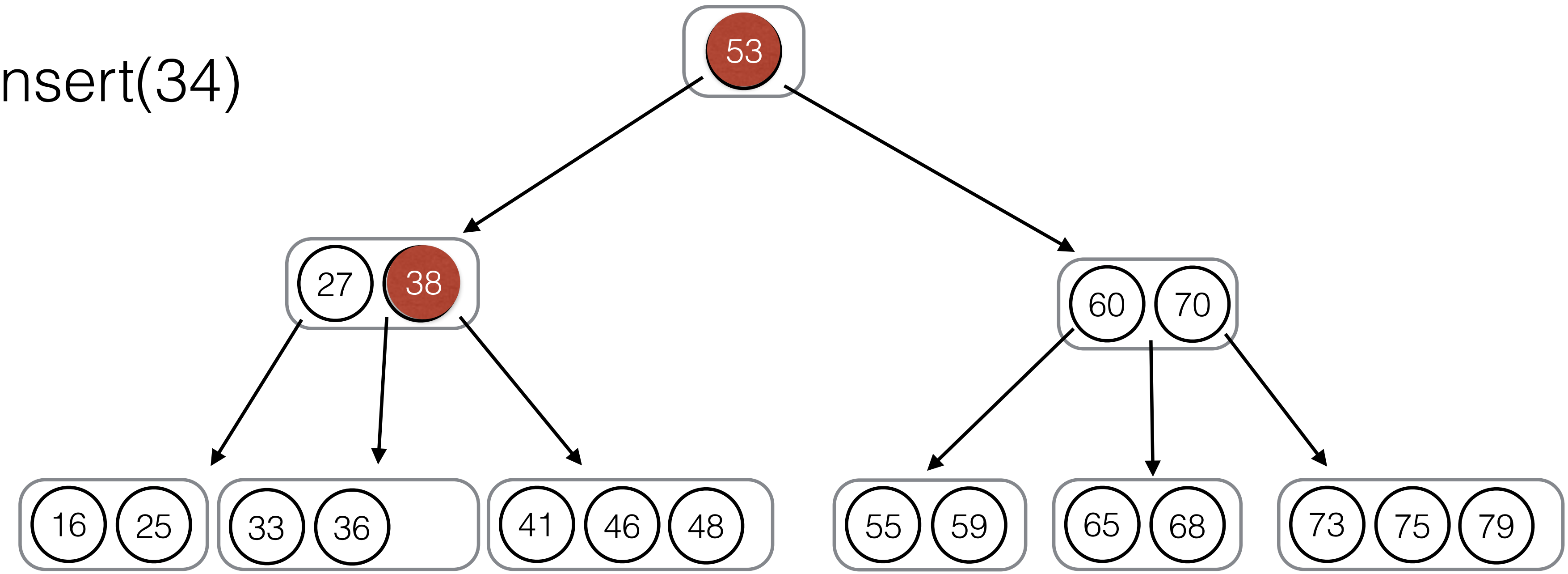
`contains` in a 2-3-4 Tree



- At each level try to find the item: 2 steps = O(c)
- If not found, follow reference down the tree. There are at most O(height(T)) = O(log N) references.

`insert` into a 2-3-4 Tree

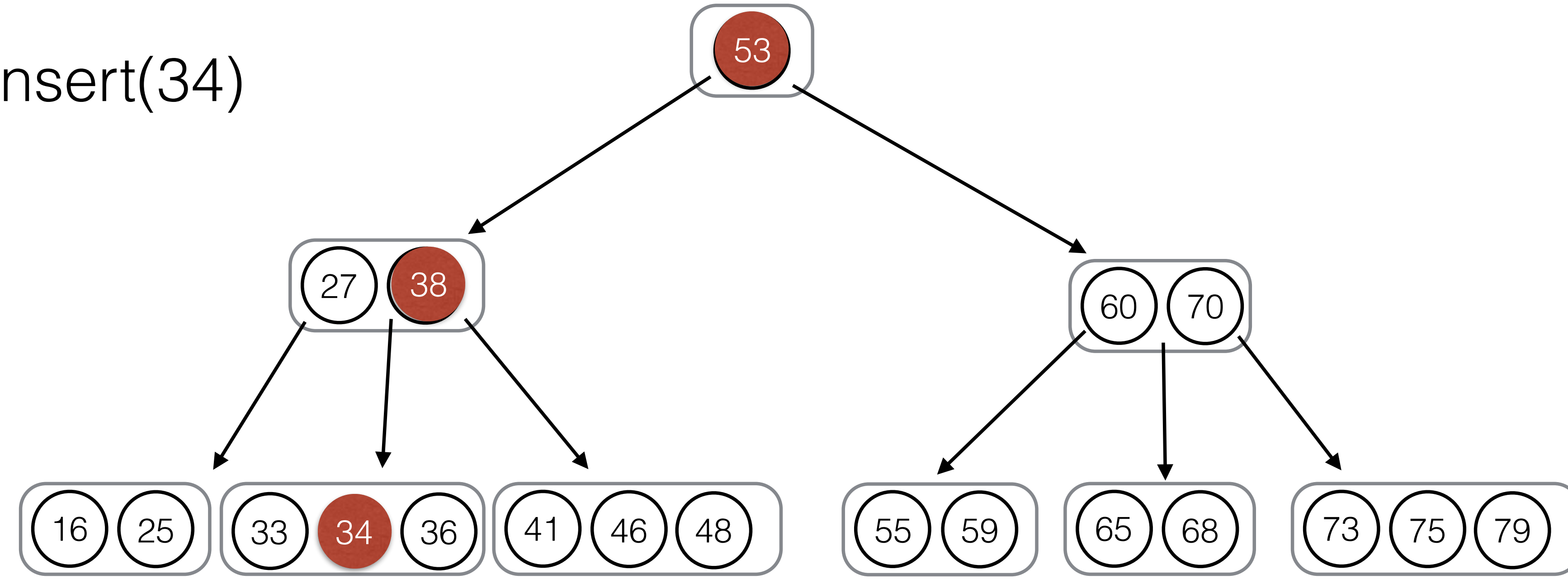
insert(34)



- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.

`insert` into a 2-3-4 Tree

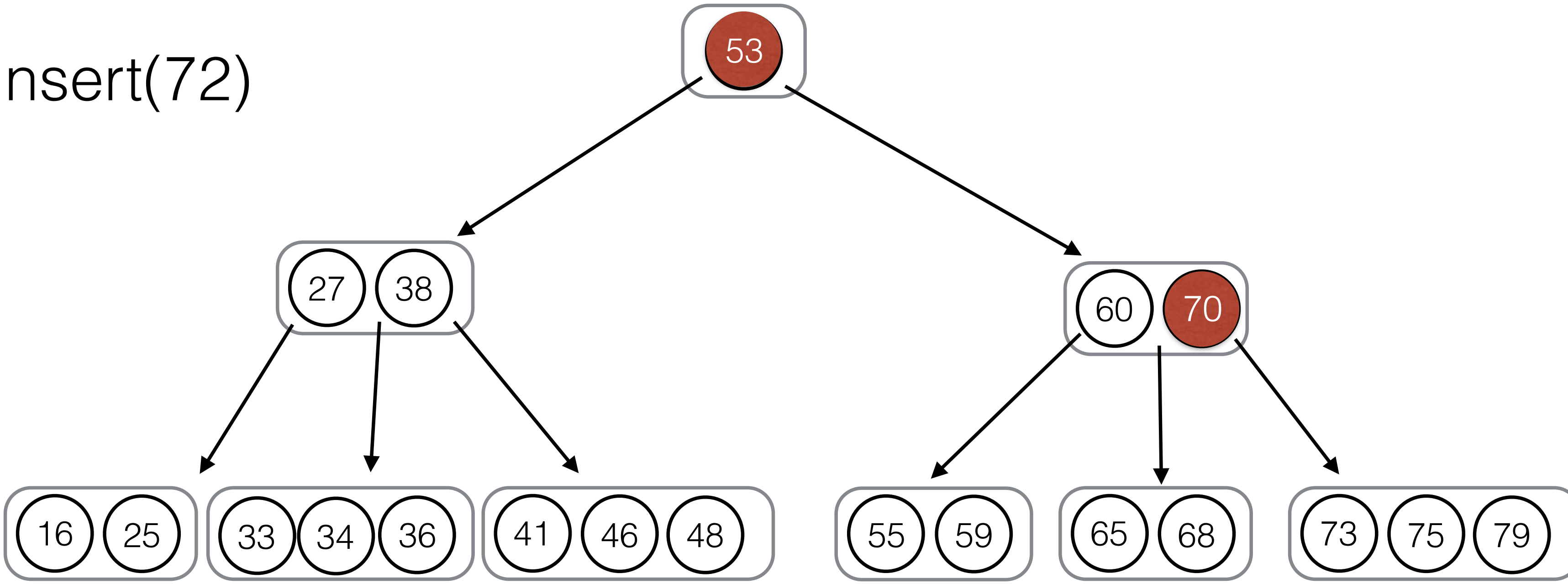
insert(34)



- Follow the same steps as contains.
- If X is found, do nothing.
- If there is still space in the leaf that should contain X, add it.
- **What if the leaf is full?**

`insert`: splitting nodes

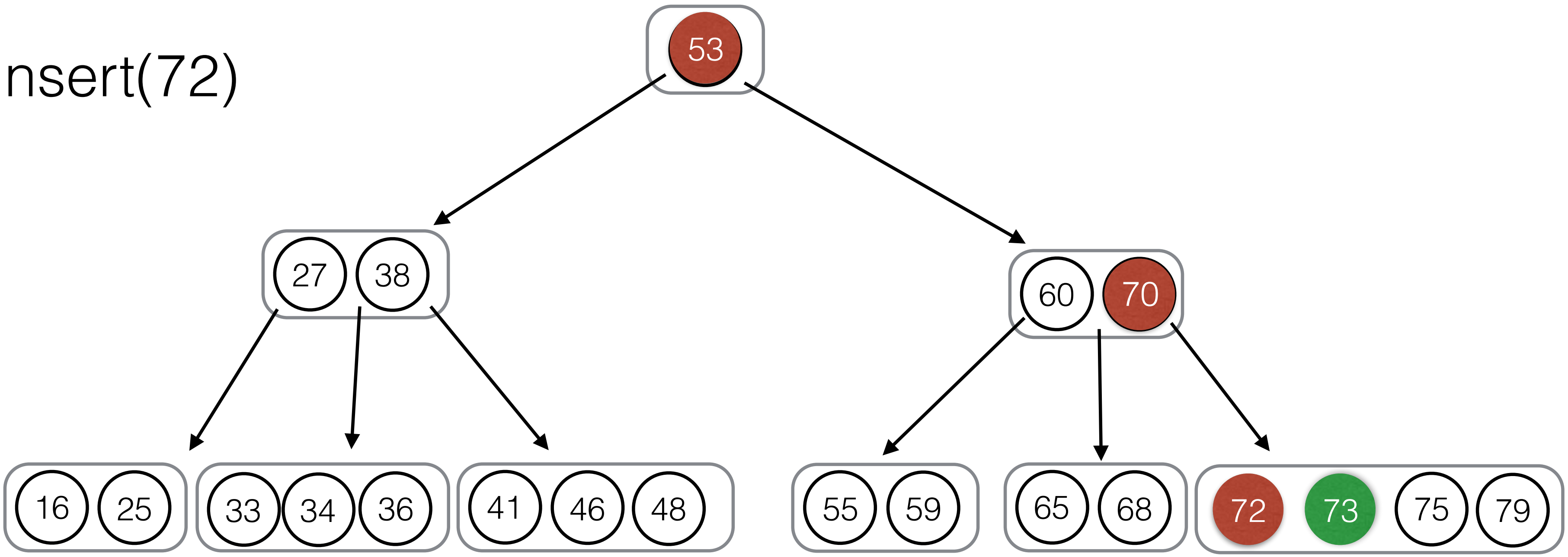
insert(72)



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

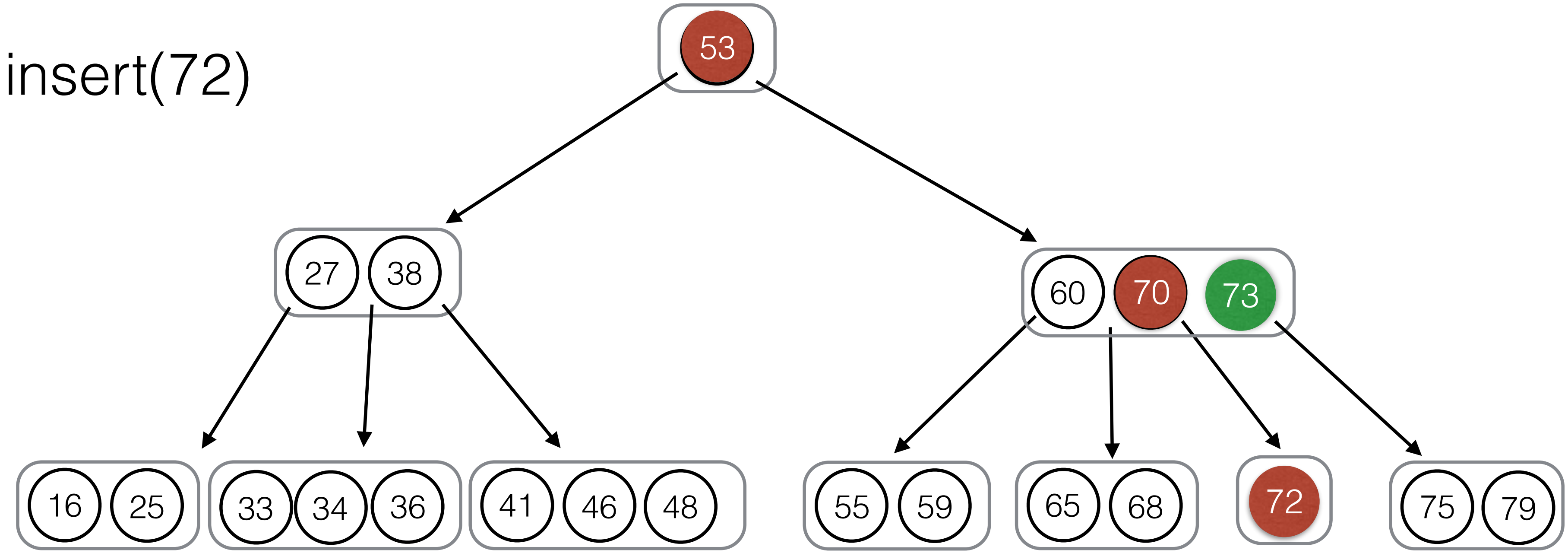
`insert`: splitting nodes

insert(72)



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

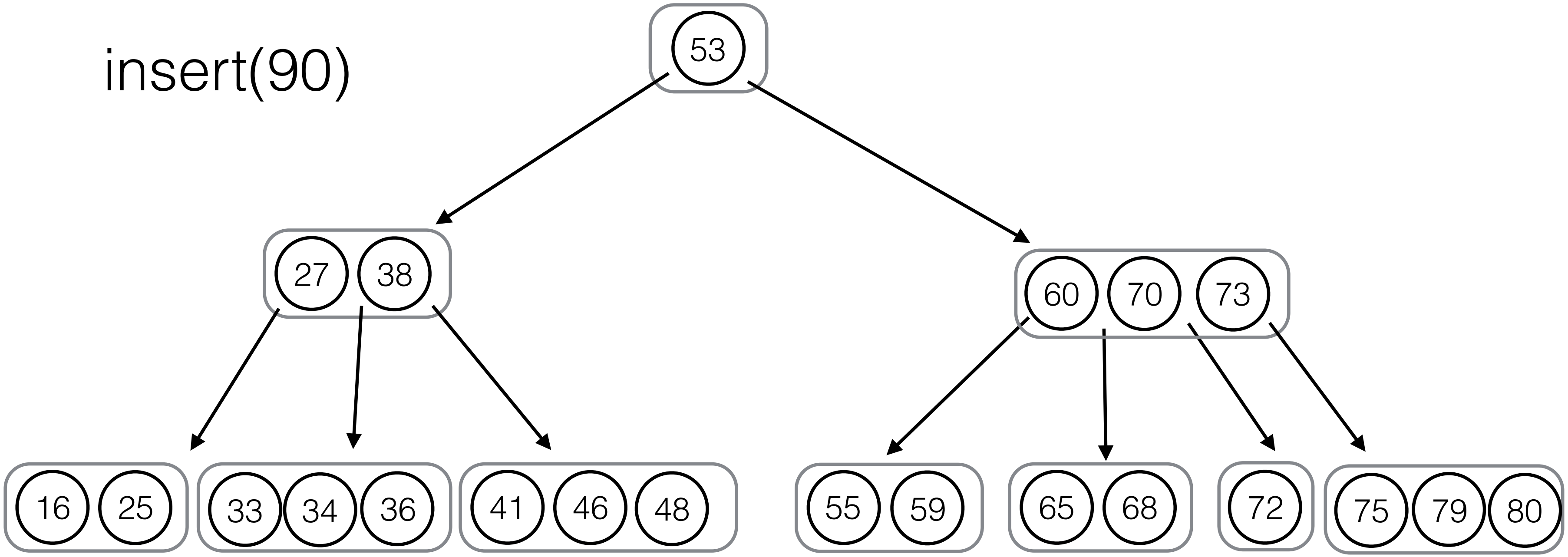
`insert`: splitting nodes



- If the leaf is full, evenly split it into two nodes.
  - choose median *m* of values.
  - left node contains items < *m*, right node contains items >*m*.
  - add median items to parent, keep references to new nodes left and right of it.

`insert:` splitting nodes

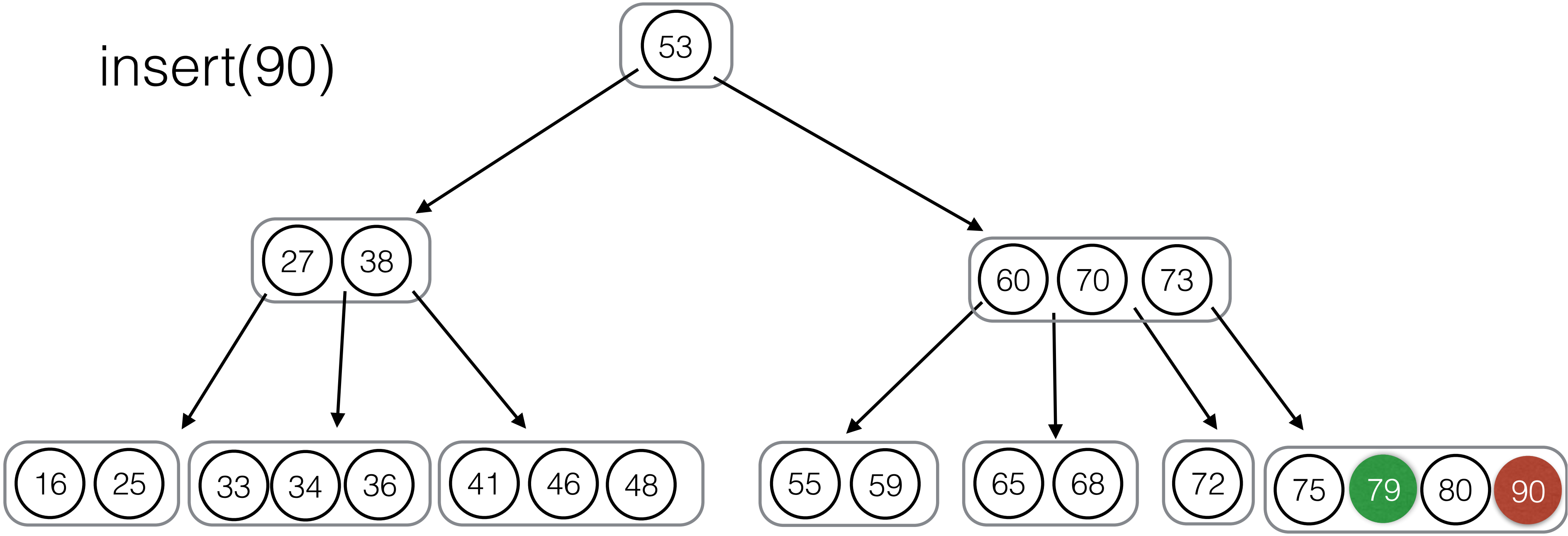
insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with **splitting old root as two children**
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

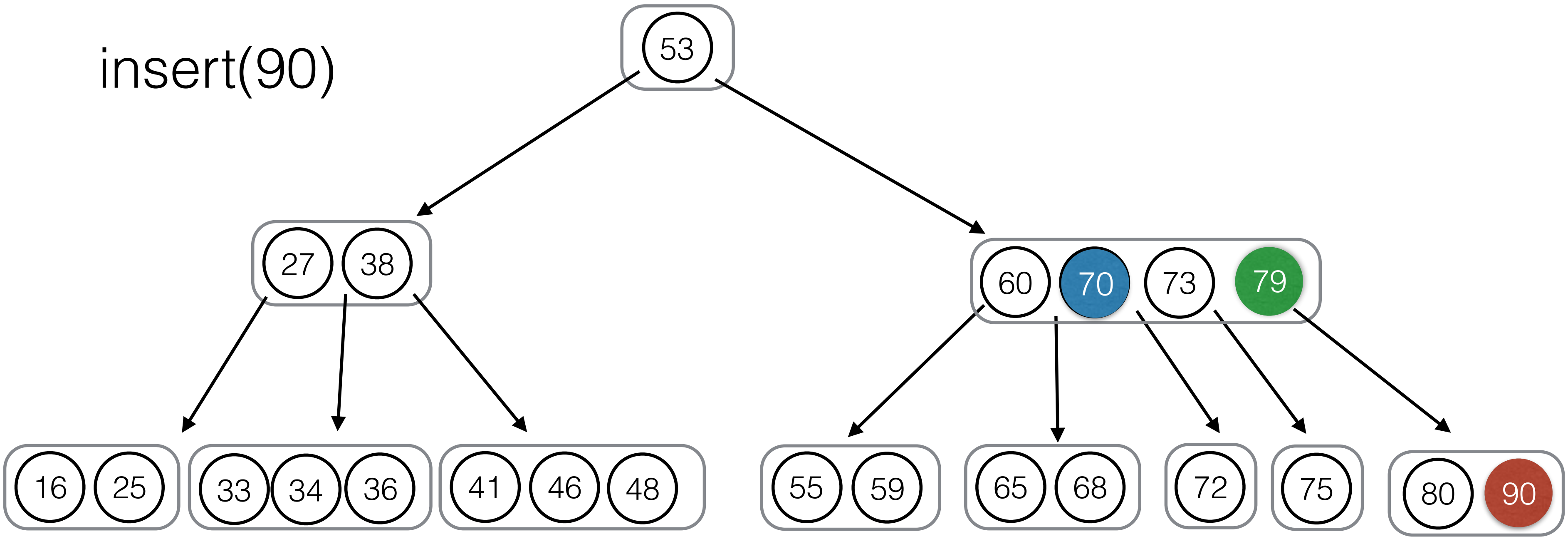
`insert`: splitting nodes

insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with **splitting old root as two children**
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

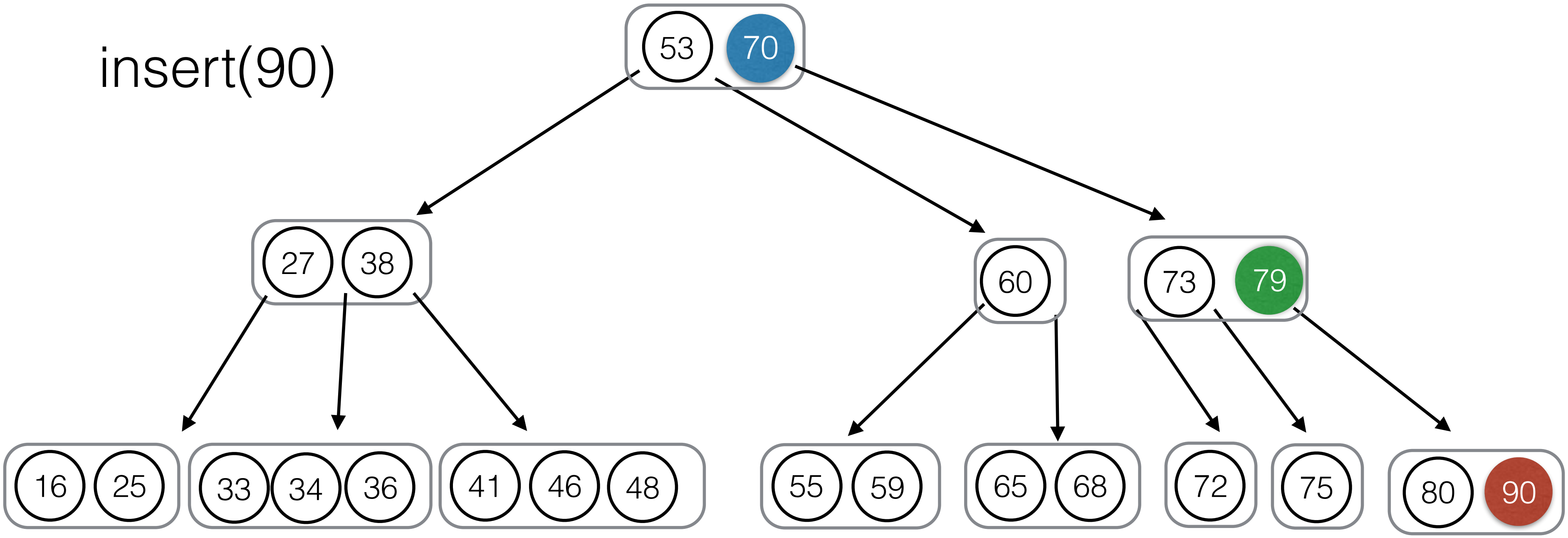
`insert`: splitting nodes



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with **splitting old root as two children**
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

`insert`: splitting nodes

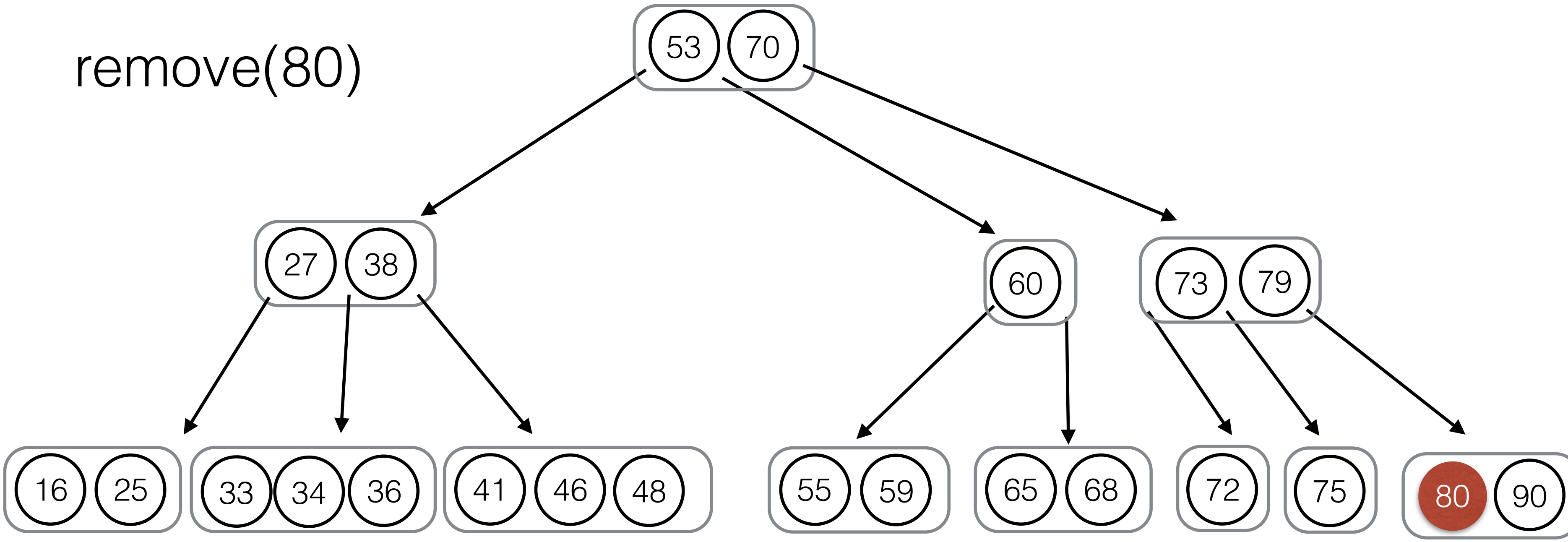
insert(90)



- If parent is also full, continue to split the parent until space can be found.
- If root is full, create a new root with **splitting old root as two children**
- At most we need one pass down the tree and one pass up, so insertion is O(log N).

`remove` from a 2-3-4 tree

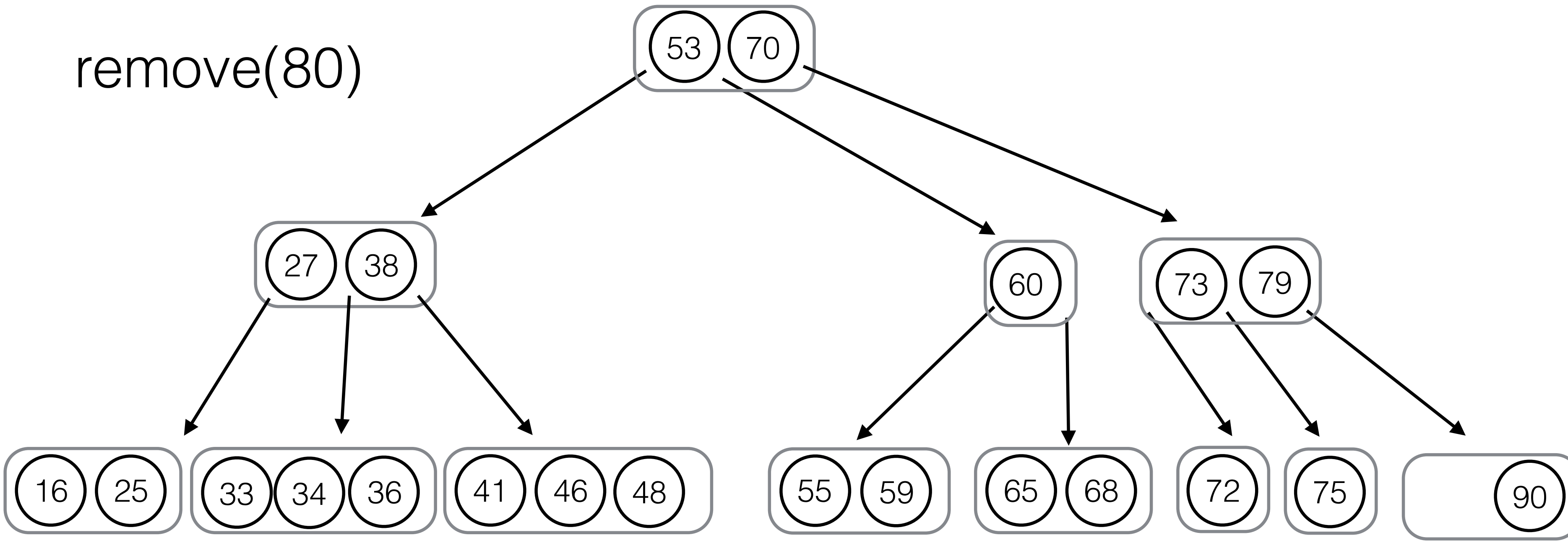
remove(80)



- Item in a 3- or 4-leaf can just be removed.

`remove` from a 2-3-4 tree

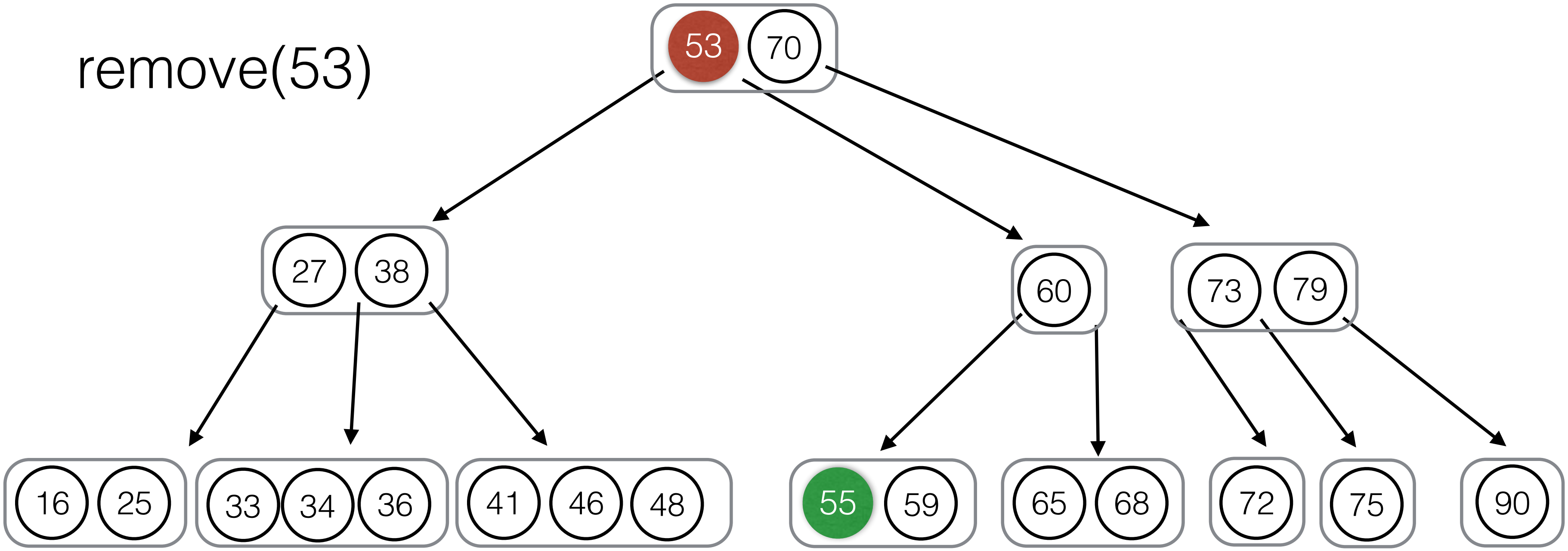
remove(80)



- Item in a 3- or 4-leaf can just be removed.

`remove` from a 2-3-4 tree

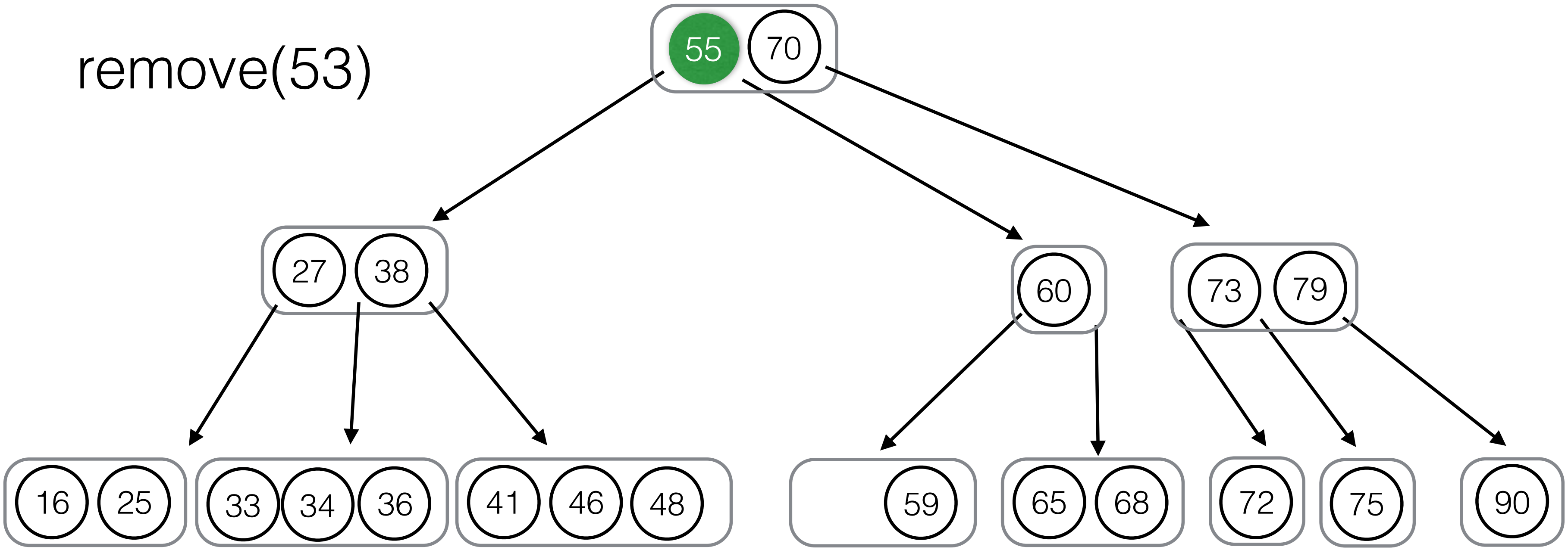
remove(53)



- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

`remove` from a 2-3-4 tree

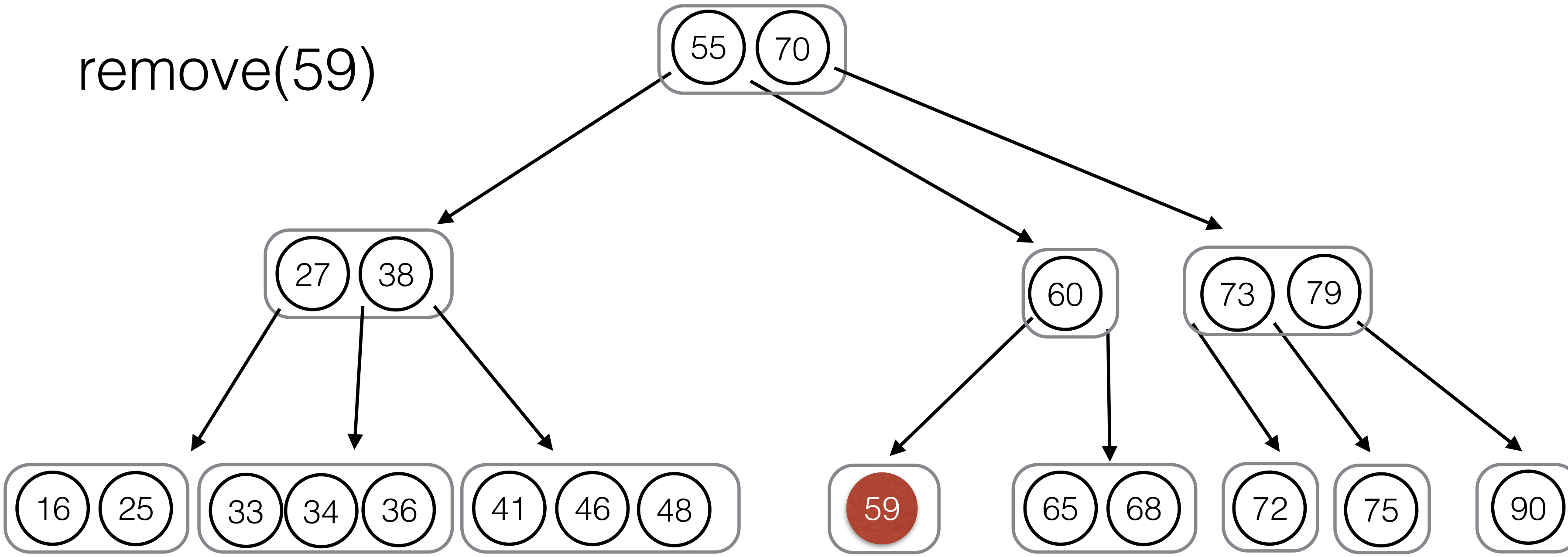
remove(53)



- Removal of an item v from internal node:
  - Continue down the tree to find the leaf with the next highest item *w*. Replace *v* with *w*. Remove *w* from its original position recursively.

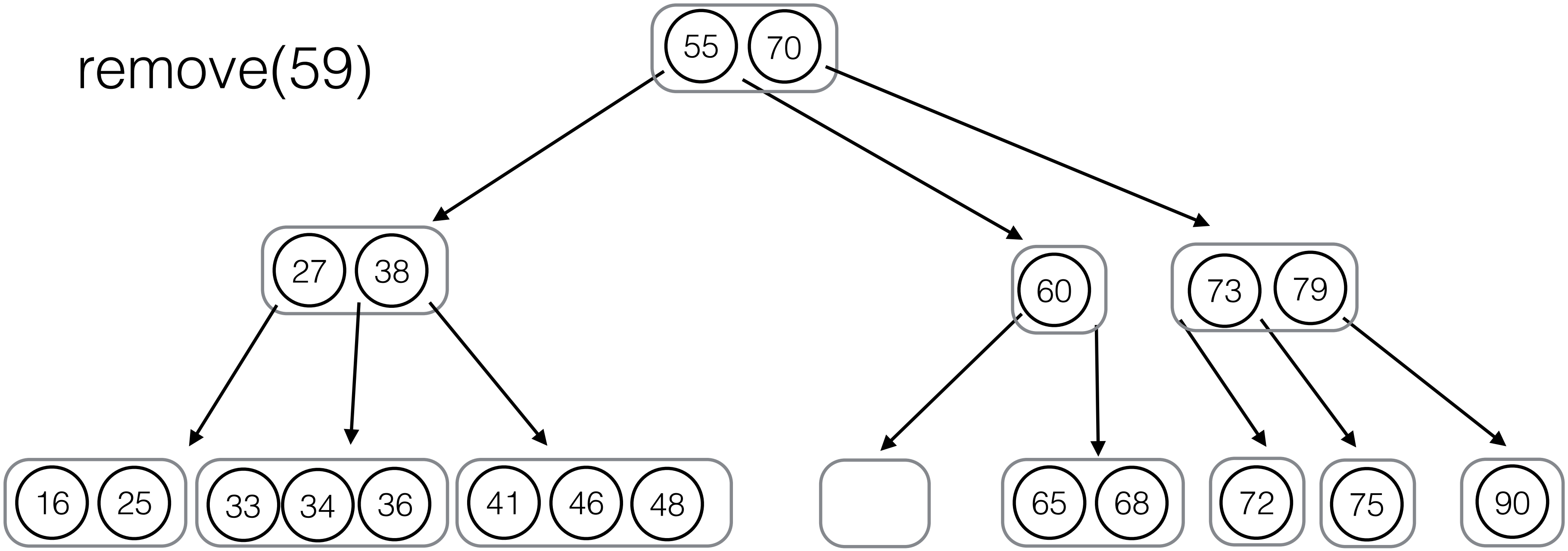
`remove` from a 2-3-4 tree

remove(59)



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

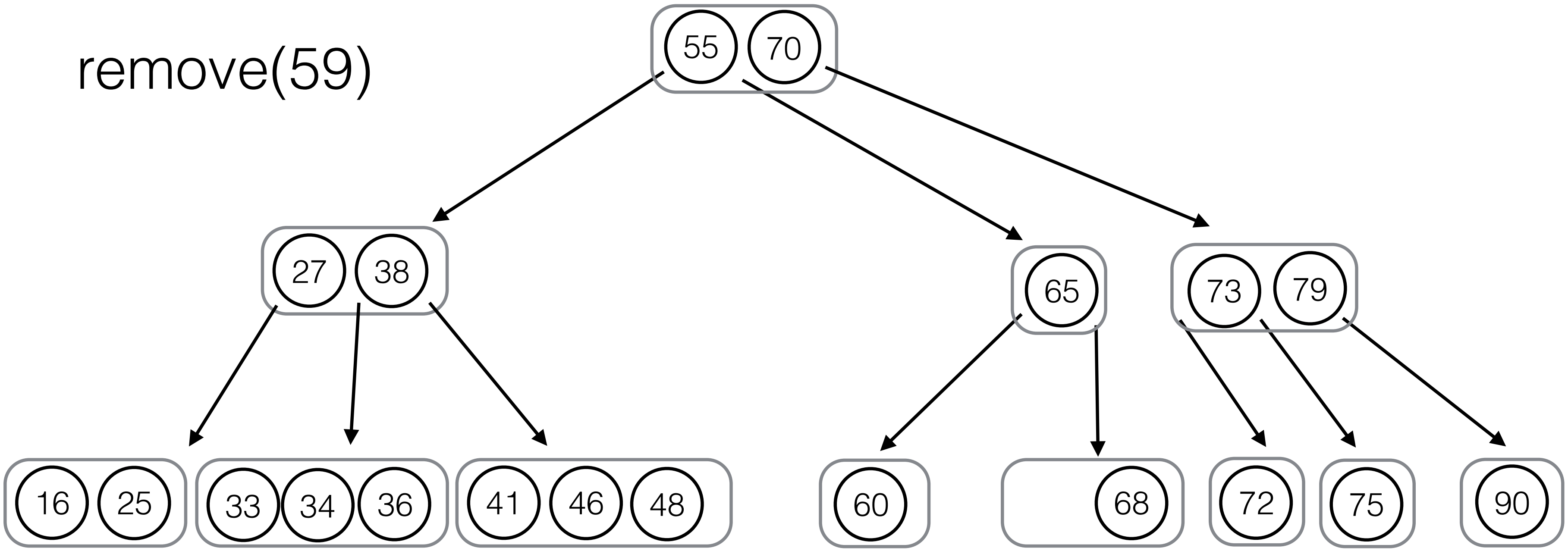
`remove` from a 2-3-4 tree



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

`remove` from a 2-3-4 tree

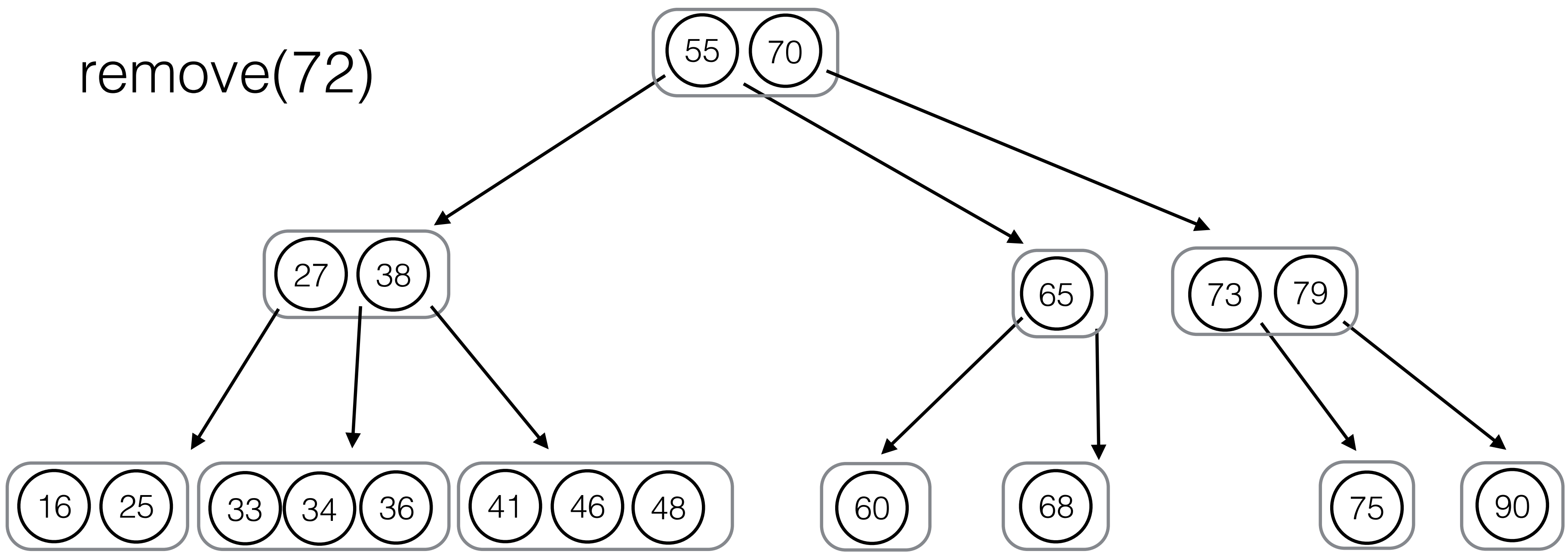
remove(59)



- Removal of an item form a leaf 2-node *t:*
  - We cannot simply remove *t* because the parent would not be well formed.
  - Move down an item from the parent of t. Replenish the parent by moving item from one of *t's* siblings.

What if no sibling is a 3 or 4 node?

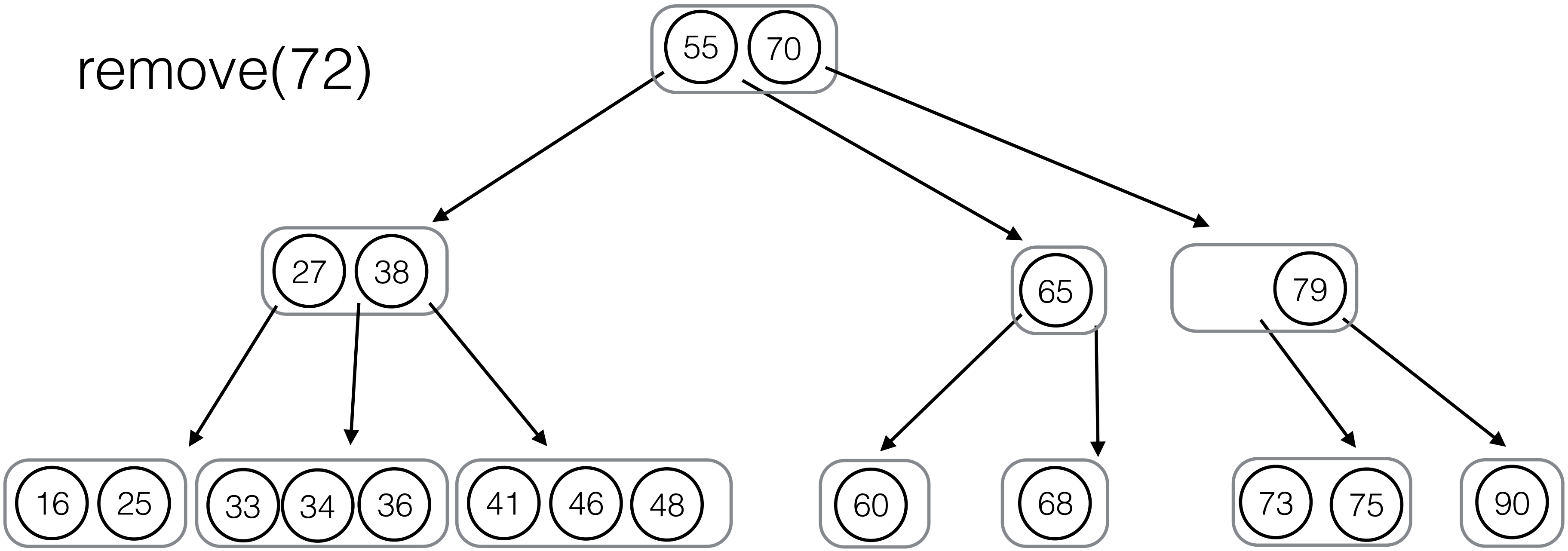
`remove` from a 2-3-4 tree



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

`remove` from a 2-3-4 tree

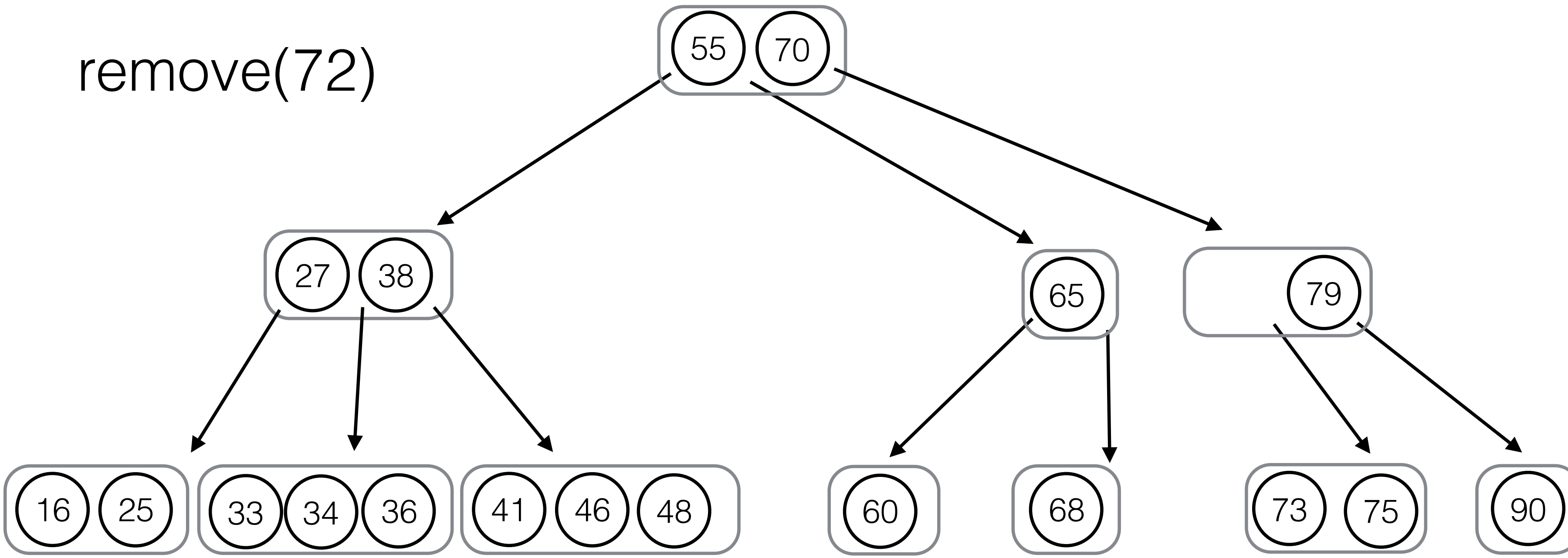
remove(72)



- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

`remove` from a 2-3-4 tree

remove(72)

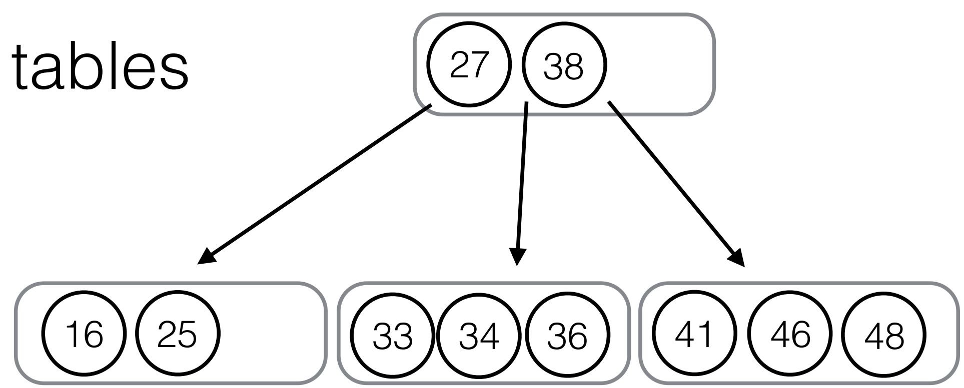


- Removal of a an item in a leaf 2-node that has no 3- or 4-node siblings:
  - **Fuse** the sibling node with one of the parent nodes.

All modifications to fix the tree are local and therefore O(c).
Remove runs in O(log N).

B-Trees

- A B-Tree is a generalization of the 2-3-4 tree to M-ary search trees.
- Every internal node (except for the root) has $\lceil \frac{M}{2} \rceil \leq d \leq M$ children and contains $d - 1$ values.
- All leaves contain $\lceil \frac{L}{2} \rceil \leq d \leq L$ values (usually L=M-1)
- All leaves have the same depth.
- Often used to store large tables on hard disk drives. (databases, file systems)



Memory Hierarchy

Typical Memory Size		Typical Access Times
< 1KB	CPU registers	5 ns
8MB	CPU caches	10 ns
64GB (or less)	Main Memory	100 ns
>500GB	Disk Storage	5 ms = 5×10^6 ns 200 accesses/second

Memory access is **much** faster than disk access.

Large BST on Disk (1)

- Assume we have a very large database table, represented as a binary search tree:
    - 10 million items, 256 bytes each.
    - 6 disk accesses per second (shared system).
- Assume no caching, every lookup requires disk access.

Large BST on Disk (2)

- Disk access time for finding a node in an unbalanced BST:

  - depth of searched node is N in the **worst case**:
    - 10 million items -> 10 million disk accesses
    - 10 million / 6 accesses per second ≈ 19 days!

  - **Expected** depth is *1.38 log N*
    - $1.38 \log_2 10 \times 10^6$ *items ≈ 32 disk accesses*
    - *32 / 6 accesses per second ≈ 5 seconds*

Large BST on Disk (2)

- Even for AVL Tree the worst case and average case will be around log N.
- *A*bout 24 disk accesses in 4 sec.

Estimating the ideal M for a B-Tree

(M-1)*256 bytes

...

M * 8 bytes

- Assume 8KB= 8,192 byte block size.
- Every data item is 256 byte.
- An M-ary B-Tree contains at most *M-1* data items + M block addresses of other trees (a 8 byte pointer each).
- How big can we make the nodes?

$$(M-1)\cdot \mathbf{256\ byte} + M\cdot \mathbf{8\ byte} = \mathbf{8,192\ byte}$$

$$M = \mathbf{32}$$

Calculating Access Time

- We representing 10,000,000 items in a B-Tree with M=32

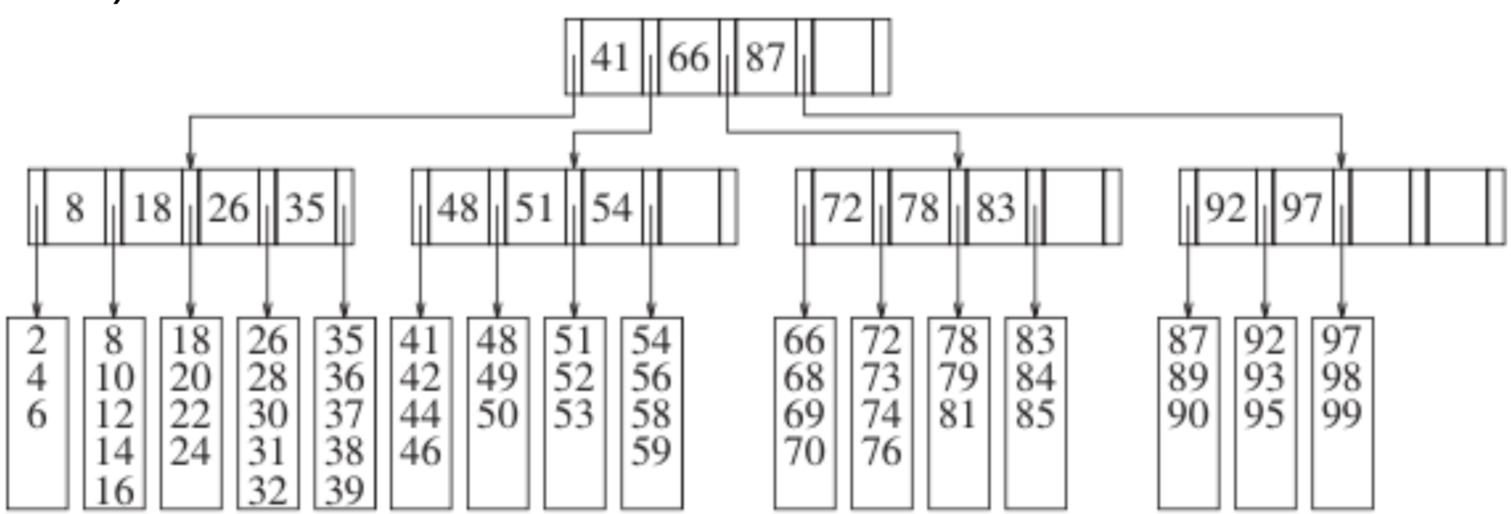
- The tree has a worst-case height of $log_{\frac{M}{2}} N$

$$log_{\frac{32}{2}} 10,000,000 \approx 6$$

- Worst-case time to find an item is
  6 accesses / 6 disk accesses per second = 1 *second*

B+ Trees

- Only leafs store full (key, value) pairs.
- Internal nodes only contain keys to help find the right leaf.
- Insert/removal only at leafs (slightly simpler, see book).



B^+ Trees on Disk

- Assume keys are 32 bytes.

$$(M-1)\cdot \mathbf{32\ byte} + M \cdot \mathbf{8\ byte} = \mathbf{8,192\ byte}$$

- We can fit at most M=205 keys in each node.

- Worst case time for 1 million keys:

$$\mathbf{log}_{\frac{205}{2}}\ \mathbf{10,000,000 = 3}$$

  - 3 accesses / 6 seconds per access = .5 seconds

Balanced Search Trees (II)

- Red-black trees
- B & B+ trees
- Take-home messages

B+ Trees

【Definition】A B+ tree of order M is a tree with the following structural properties:

(1) The root is either a leaf or has between 2 and M children.

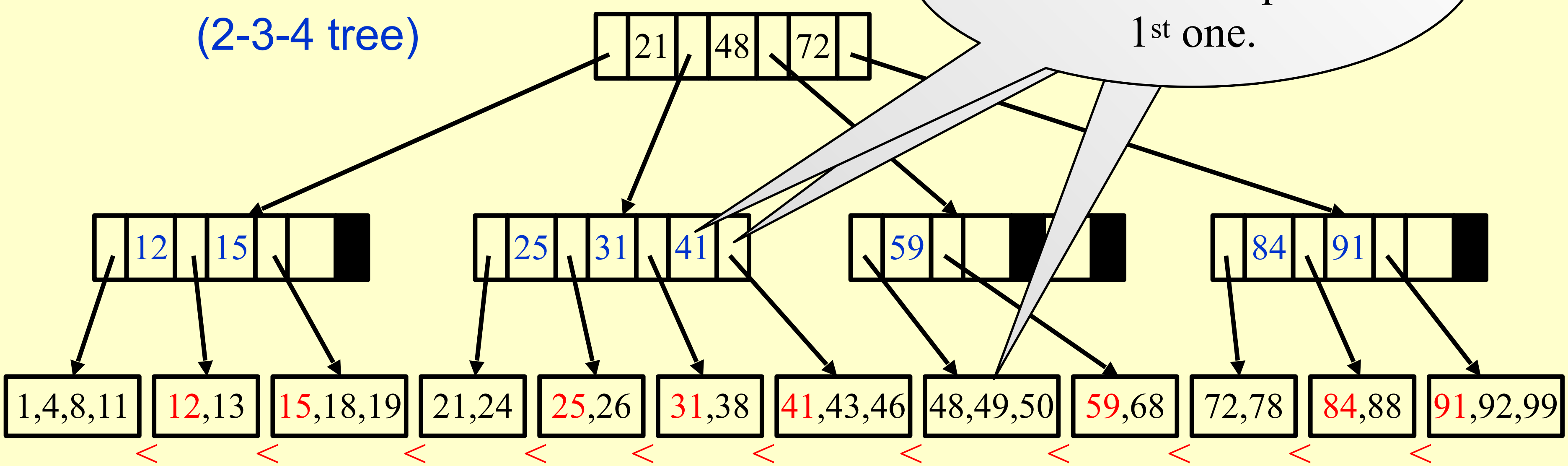
(2) All nonleaf nodes (except the root) have between ⌈M/2⌉ and M children.

(3) All leaves are at the same depth.

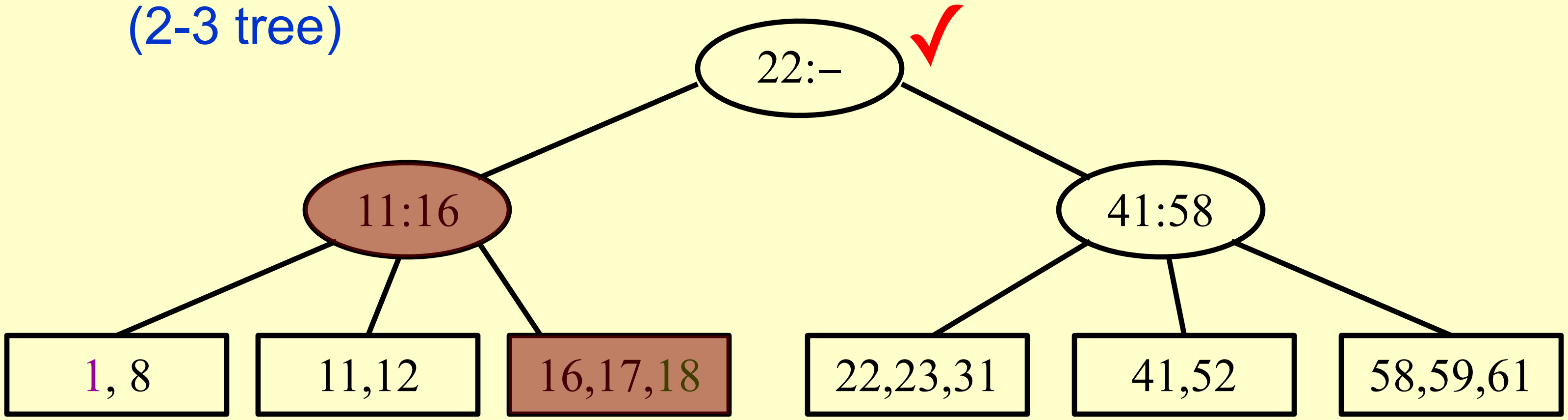
Assume each nonroot leaf also has between ⌈M/2⌉ and

And $M - 1$ smallest key values in the subtrees except the 1st one.

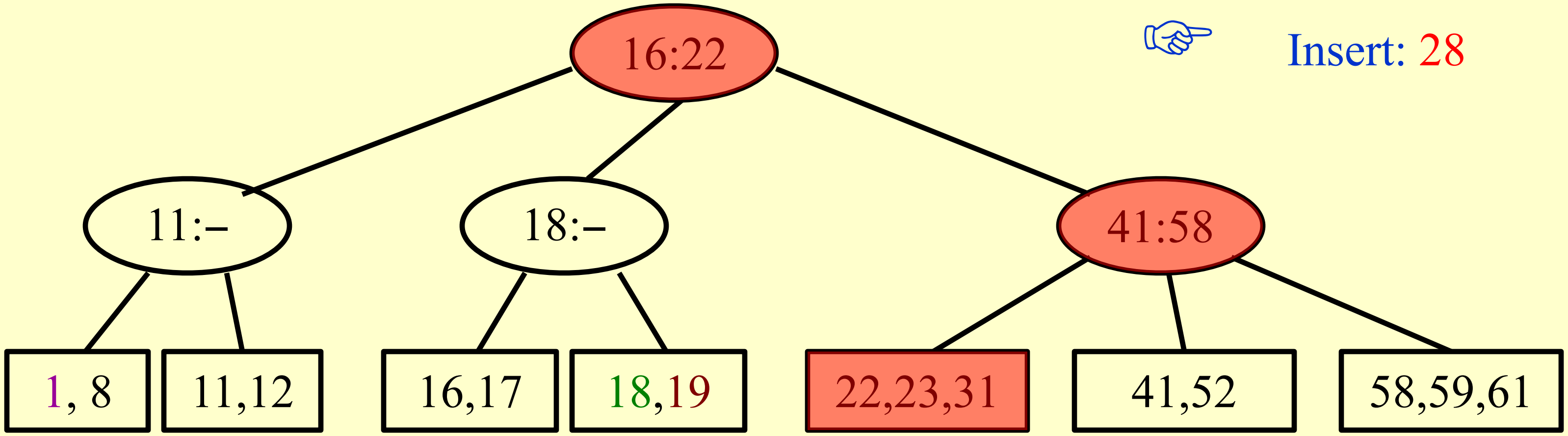
A B+ tree of order 4
(2-3-4 tree)



A B+ tree of order 3
(2-3 tree)

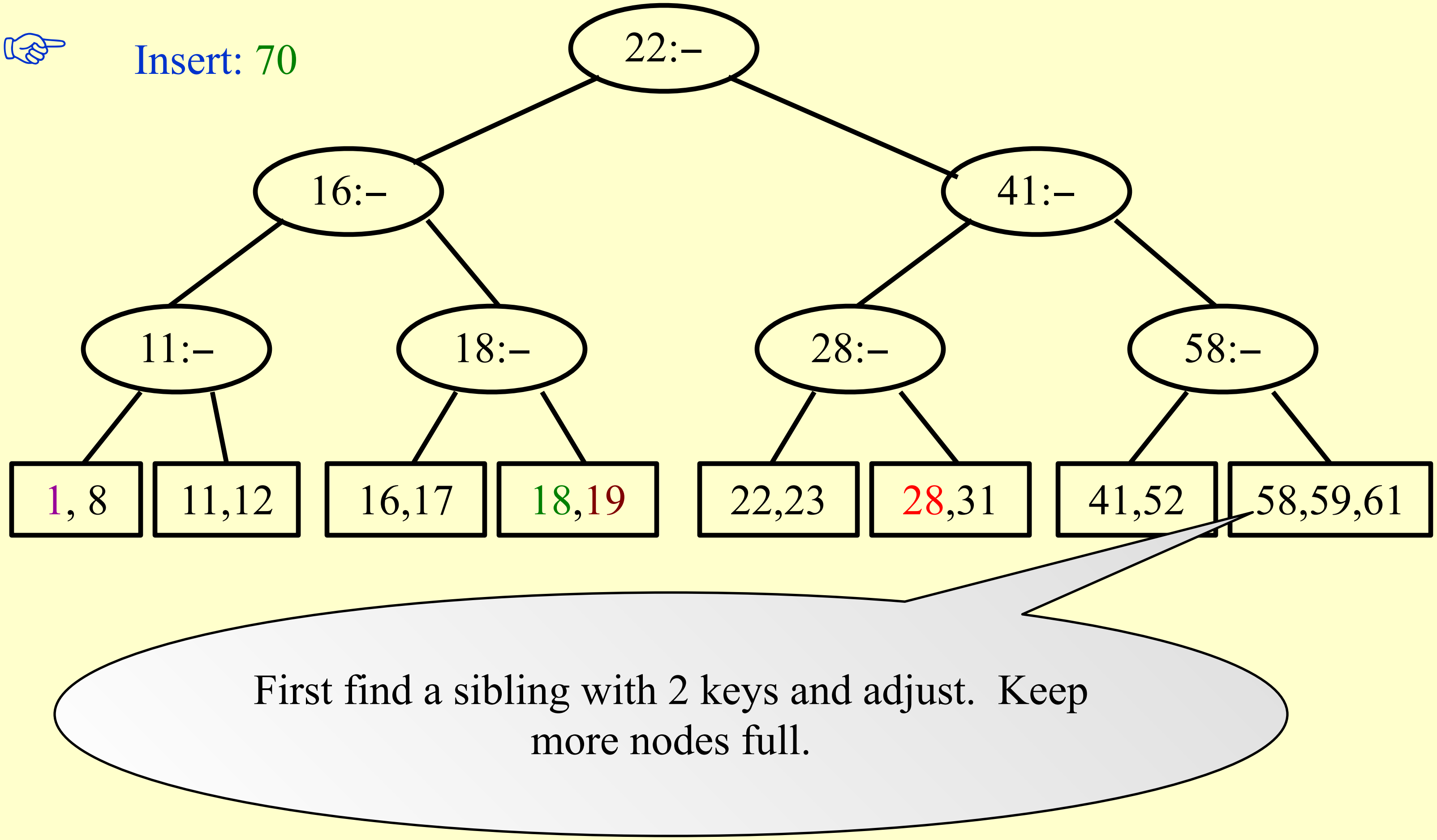


☞ Find: 52 ☞ Insert: 18 ☞ Insert: 1 ☞ Insert: 19



☞ Insert: 28

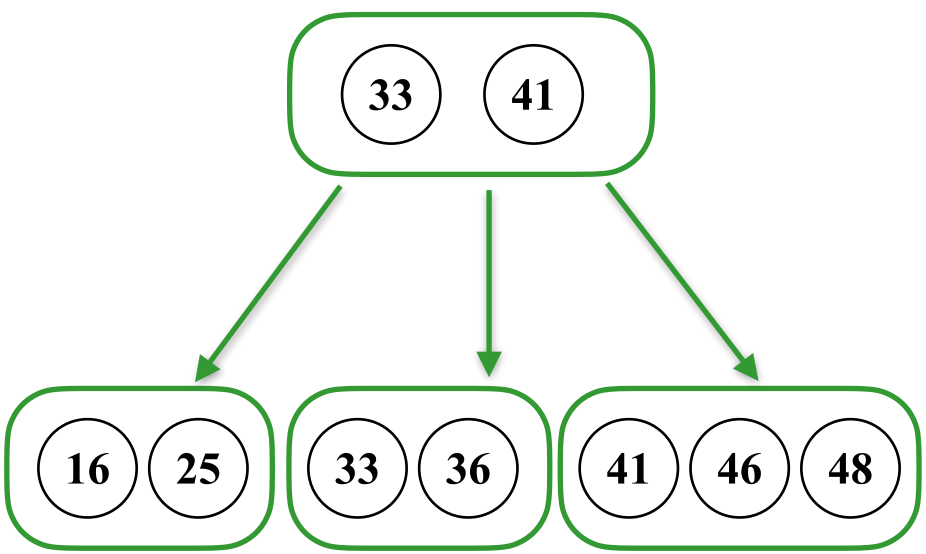
☞ Insert: 70



☞ Deletion is similar to insertion except that the root is removed when it loses two children.

Deletion of B+ Tree

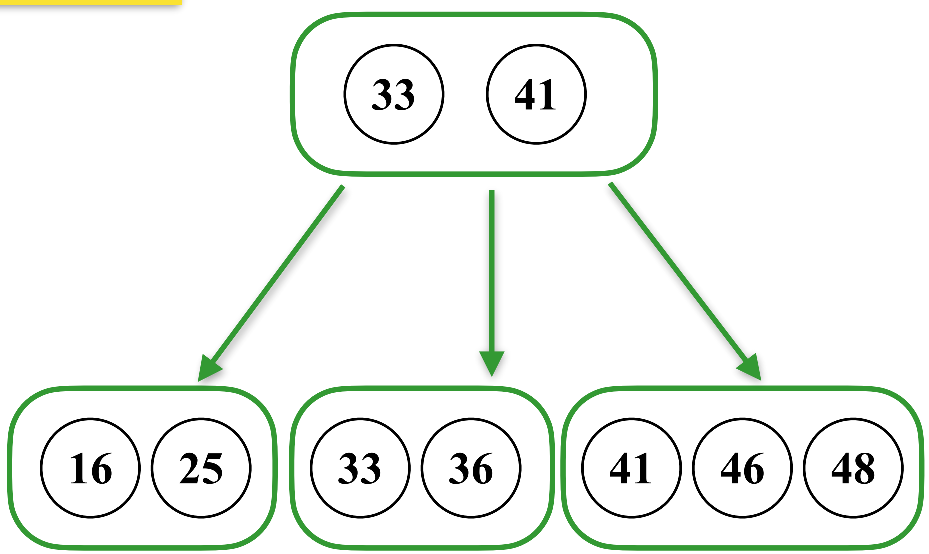
In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node



Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

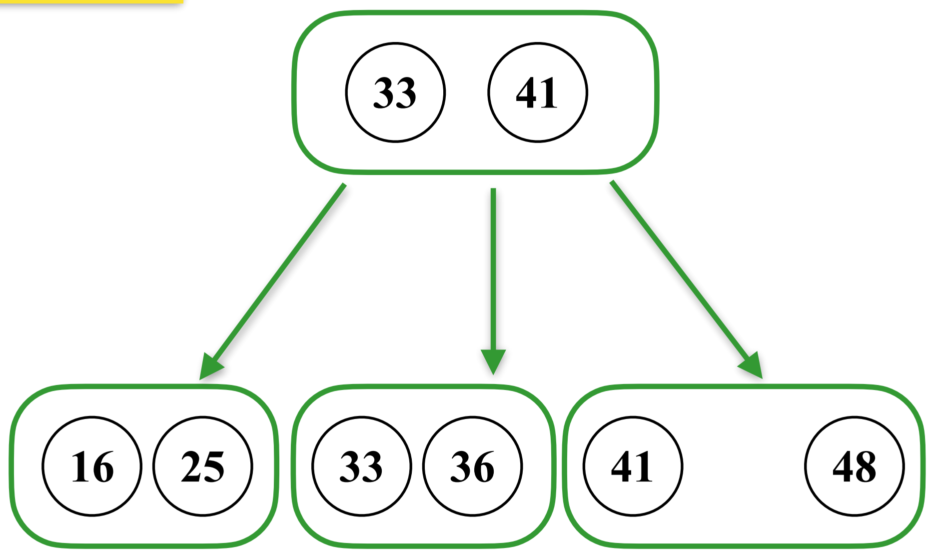
case 1: Delete(46)



Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

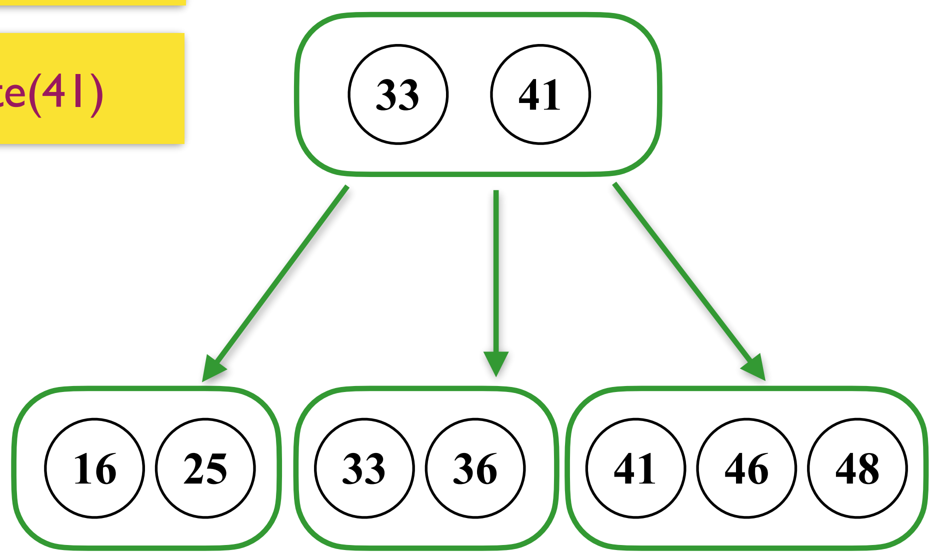


Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)



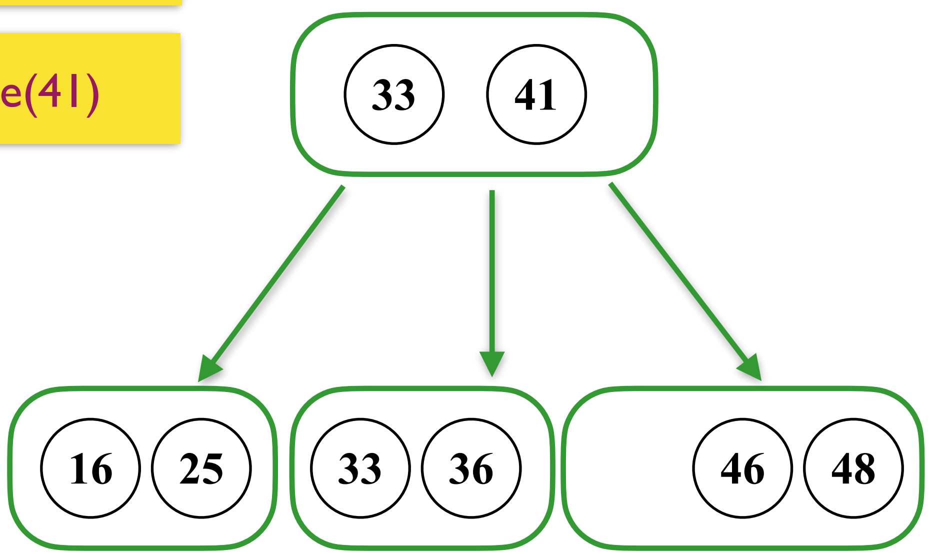
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!

Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

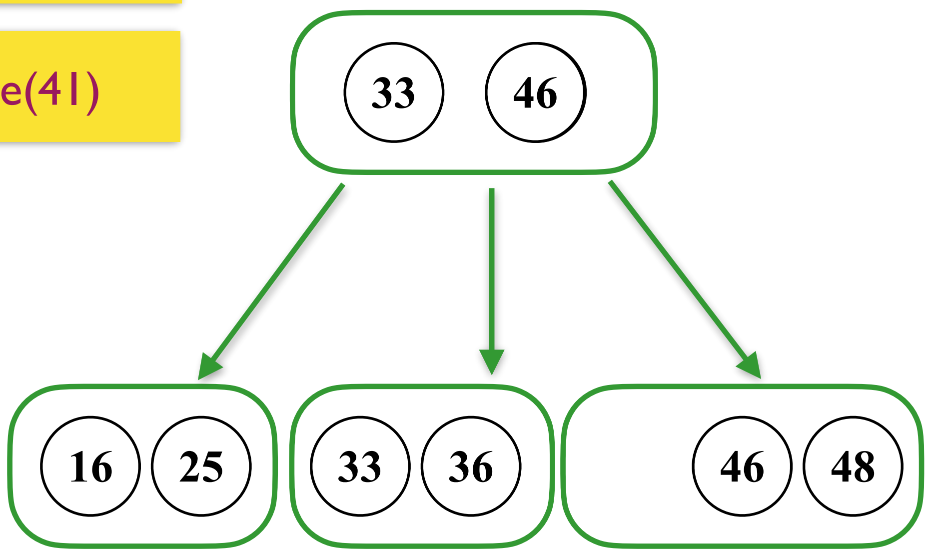


Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

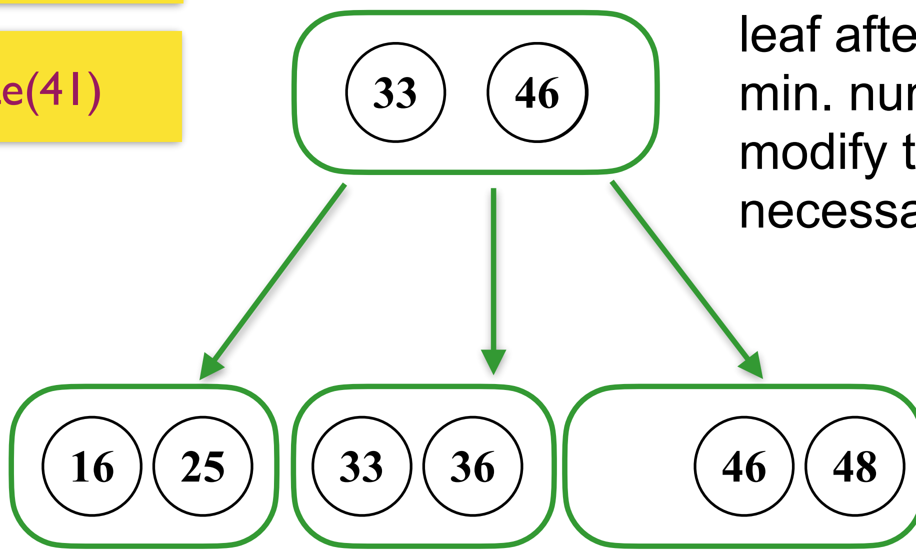


Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)



if the number of keys in the leaf after deletion is beyond min. number, just remove or modify the parent key if necessary

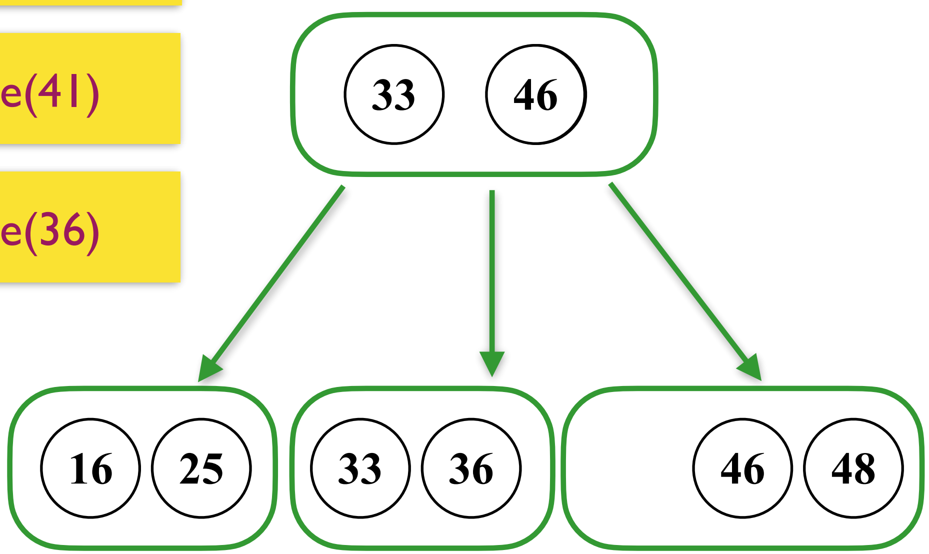
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



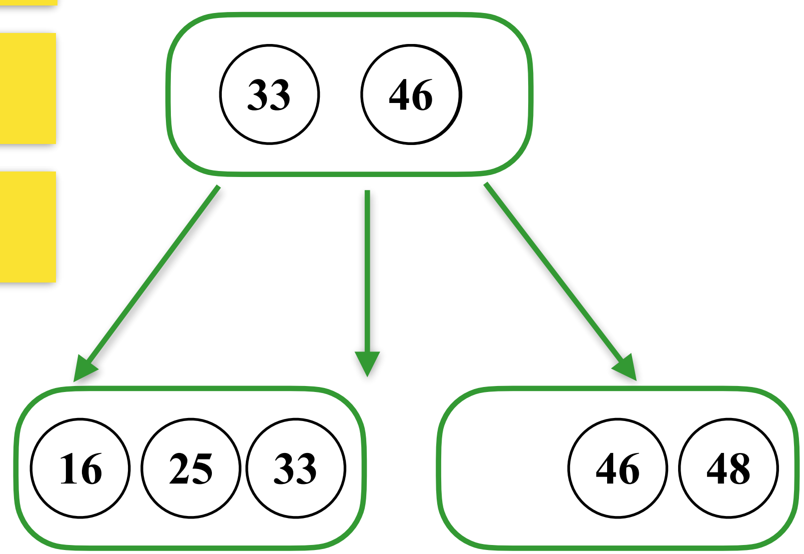
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



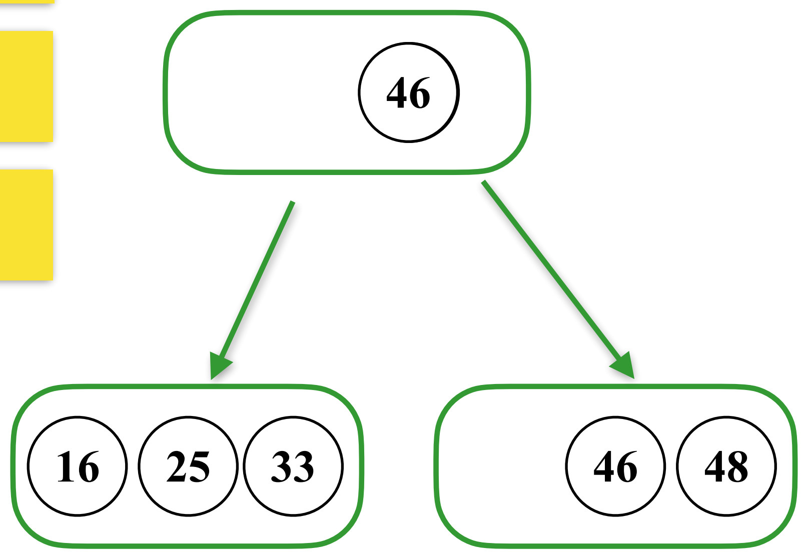
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



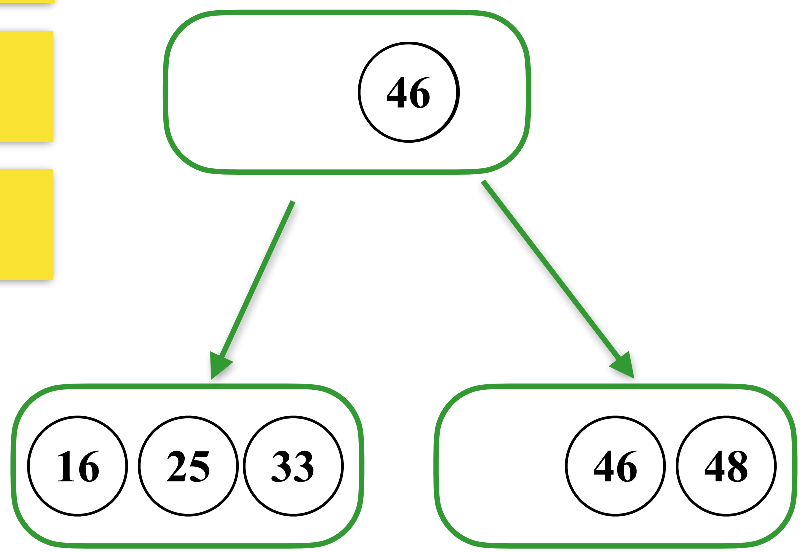
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



case 4: Delete(48)

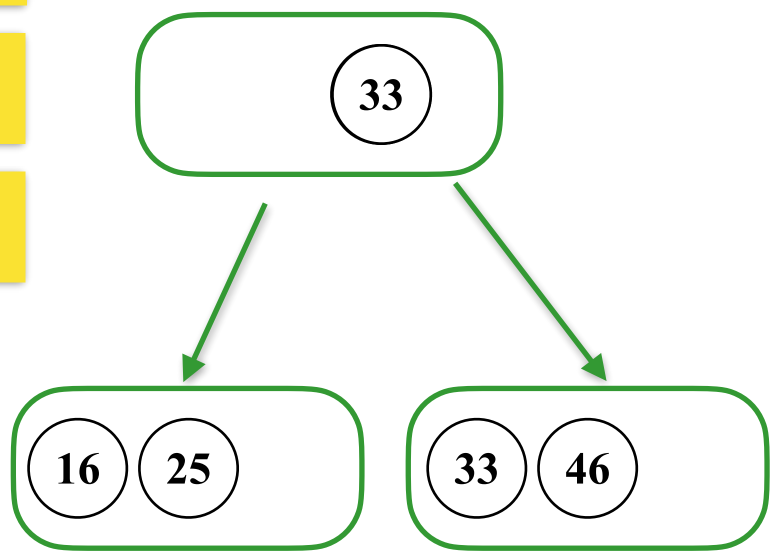
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



case 4: Delete(48)

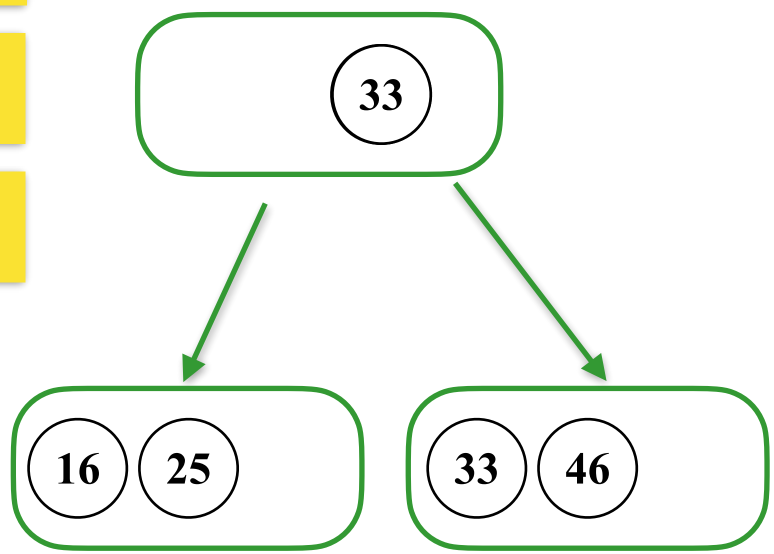
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



case 4: Delete(48)

case 5: Delete(33)

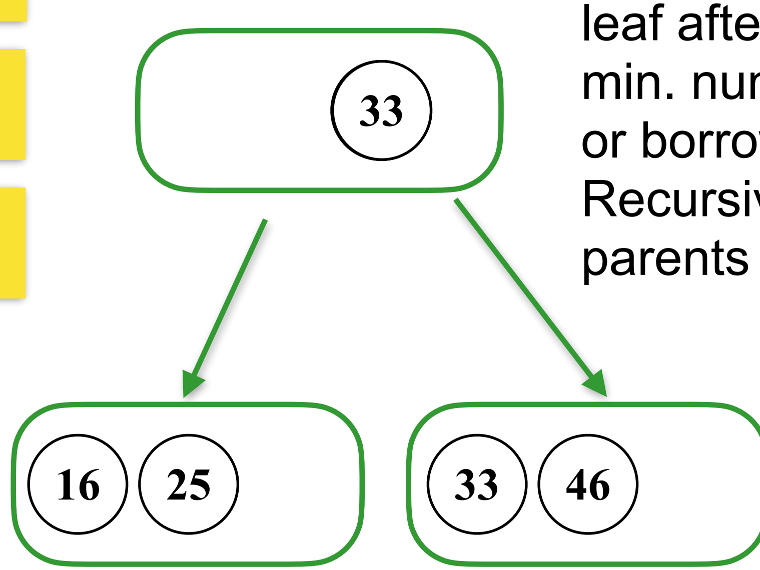
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



if the number of keys in the leaf after deletion is below min. number, try to merge or borrow from siblings. Recursively delete key from parents if necessary.

case 4: Delete(48) **case 5:** Delete(33)

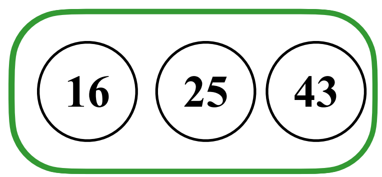
Deletion of B+ Tree

In all homework and exams, only B+ tree is considered.
All 2-3 and 2-3-4 trees in HW and exams are B+ trees!
Assumption in our course:
max(min) number of keys for leaf node = max(min) number of children for non-leaf node

case 1: Delete(46)

case 2: Delete(41)

case 3: Delete(36)



if the number of keys in the leaf after deletion is below min. number, try to merge or borrow from siblings. Recursively delete key from parents if necessary.

case 4: Delete(48)

case 5: Delete(33)

For a general B+ tree of order M

```
Btree  Insert ( ElementType X,  Btree T )
{
    Search from root to leaf for X and find the proper leaf node;
    Insert X;
    while ( this node has M+1 keys ) {
            split it into 2 nodes with ⌈(M+1)/2⌉ and ⌊(M+1)/2 ⌋ keys,
    respectively;
            if (this node is the root)
                    create a new root with two children;
            check its parent;
    }
}
```

$T = O(M)$

$T(M, N) = O(\ (M/\log M) \log N\ )$

Depth$(M, N) =$ $O(\ \lceil \log_{\lceil M/2 \rceil} N \rceil\ )$

$T_{Find}\ (M, N) = O(\ \log N\ )$

Historial Notes

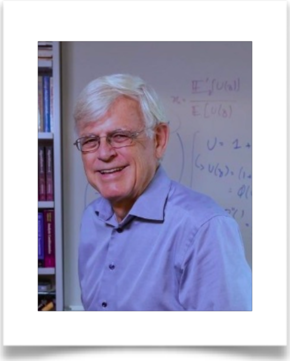
- 2-3-4 tree (1972) and B-tree (1970):

Edward M. McCreight

Rudolf Bayer

- Red-black tree (1978):

Leonidas J. Guibas



Robert Sedgewick

- 2-3 tree (1970):

John Hopcroft

Balanced Search Trees (II)

- Red-black trees
- B & B+ trees
- Take-home messages

Take-Home Messages

- Red-black trees:
  - Binary search tree version of 2-3-4 trees. The red nodes are for represent >2 branches in each node.
  - The major properties lie in that the black height is balanced for each node.
  - The insertion and deletion involve constant cost on rotations.
- B & B+ trees:
  - Search trees with more branches. Suitable for reducing access cost on nodes, applications on database, secondary drives…
  - Reduce tree depth by increasing the number of branches.

Balanced Search Trees

- AVL trees: suitable when look-up costs matter most.
- Splay trees: suitable when the same items are visited repeatedly.
- Red-black trees: suitable when insertion/deletion costs matter most.
- B&B+ trees: suitable when the data are stored in blocks, and the access costs matter most.

Thanks for your attention!
Discussions?

Reference

Introduction to Algorithms (4th Edition): Chap.13, 18.

Algorithms (4th Edition): Chap. 3.3.

http://www.cs.columbia.edu/~bauer/cs3134-f15/slides/w3134-1-lecture11.pdf